SAS/C® Compiler and Library 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 Compiler and Library User's Guide, Release 7.00, Cary, NC: SAS Institute Inc., 2001.

SAS/C Compiler and Library User's Guide, Release 7.00

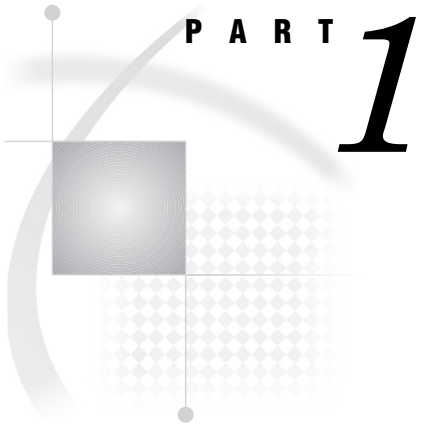
ISBN 1–58025–727–5

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.

Contents



PART 1

User's Guide

CHAPTER

1

Introduction

Introduction to the SAS/C Compiler

The SAS/C Compiler, Release 7.00 is a portable implementation of the high-level C programming language. The primary elements of the SAS/C Compiler are the compiler and the run-time library. Additionally, the compiler product also includes a number of utility programs, as well as several configurations of the run-time library for specialized environments. Although the compiler is heavily oriented for use in large software systems, it is an efficient tool for any software project that is written in the C language.

Compiler features include the following:

- generation of reentrant code, enabling many users to share the same code.
- an optimization phase to increase speed and efficiency of generated code.
- the ability for generated code to be executed in both 24-bit and 31-bit addressing modes, allowing compiled programs to execute above the 16-megabyte line under any extended architecture operating system.
- identical generated code for OS/390 and CMS operating systems, allowing compatibility without recompiling.
- built-in functions, including many of the traditional string-handling functions, that generate inline machine code rather than function calls.
- support for low-level systems programming through inline machine code and the Systems Programming Environment (SPE).
- full support for interlanguage communication.
- dynamic loading of I/O support routines. Modules are loaded as needed at run time.
- Full-Screen Support Library (FSSL) (optional).

- full-function debugger.
- BSD socket support.

The compiler also provides a complete library of functions, including functions compatible with the International Standards Organization (ISO) and American National Standards Institute (ANSI) C language standards as well as functions that support nonstandard features such as interuser communication. The library also includes functions defined by the ISO POSIX 1003.1 and 1002.1a standards, plus other functions from UNIX operating systems. Extensive information about the library and each library function is provided in SAS/C Library Reference, Volume 1 and SAS/C Library Reference, Volume 2.

Development and Execution Environments

The SAS/C Compiler supports a number of development and execution environments that facilitate the efficient development of applications designed to run on the IBM System/370 mainframe. The SAS/C C++ Development System is an add-on product that extends the capabilities of these development environments to support the C++ programming language. The following paragraphs briefly describe these development environments.

OS/390, CMS, TSO

Traditionally, SAS/C applications have been developed in one of the System/370 operating environments: TSO, CMS, or OS/390 batch. This book is designed to provide information you will need to use the SAS/C Compiler effectively in the mainframe SAS/C development environment.

XA CMS Support

There are two versions of the SAS/C Compiler available under CMS: the 370 mode version and the XA version. The 370 mode version of the compiler (and programs developed with it) runs under CMS for VM/SP Releases 3, 4, 5, 6, and later. This version runs in tolerance mode (XA-mode but only AMODE 24) under CMS for VM/XA SP Releases 1, 2, and later. The XA version works under the CMS associated with releases of VM after Release 5 of VM/SP. Under VM extended architecture systems, programs run in exploitation mode, that is, they run AMODE 31, taking advantage of storage above the 16-megabyte line.

CICS

The SAS/C CICS Command Language Translator extends the mainframe execution environment so that you can write C programs that interact with the IBM Customer Information Control System (CICS). The SAS/C Library supports the CICS run-time environment under TSO, CMS, and OS/390 batch. The SAS/C CICS Command Language Translator is documented in the SAS/C CICS User's Guide.

UNIX System Services

With the UNIX System Services (USS) subsystem from IBM, the SAS/C Library supports the operating system interface defined by the ISO POSIX 1003.1 standard.

This support extends the mainframe development environment to the USS shell under OS/390. This support is described in Chapter 15, "Developing Applications for Use with UNIX System Services OS/390," on page 331.

Cross-Platform

The SAS/C Cross-Platform Compiler and the SAS/C C++ Cross-Platform Development System are the cornerstones of a cross-platform development environment that enables you to compile mainframe SAS/C and SAS/C C++ applications on a UNIX workstation. These add-on products run on the workstation and produce prelinked output files that can be transferred to an IBM System/370 mainframe, where they can be linked to produce an executable load module.

There are several benefits to using SAS/C and C++ cross-platform software.

Reduced mainframe load
: By moving compilations off the mainframe, mainframe CPU cycles are preserved for other users. This can amount to a significant reduction in mainframe requirements, directly translating into a cost savings.

Improved source management
: Developers may take advantage of improved source management tools, as well as the UNIX hierarchical file system.

Improved build management
: Developers may take advantage of improved build management tools, such as **`make`**.

Improved compilation turnaround
: In a heavy development environment, developers often find that performing compilations locally can result in a better turnaround time.

For more information about the cross-platform development environment, refer to SAS/C Cross-Platform Compiler and C++ Development System User's Guide.

Compatibility between versions

The following points should be considered when determining compatibility between the two CMS versions of the compiler:

- The library does not support I/O to globaled OS/390 PDSs except in 370 mode.
- Programs linked using the XA CMS library will not run under VM/SP Releases 3, 4, and 5.

Quick Start to Using the SAS/C Compiler

The following procedures provide the essential information you need to get started using the SAS/C Compiler under TSO, CMS, or the USS shell. To use these procedures, you need to create a simple source file such as the following:

```
#include <stdio.h>

main()
{
  printf("Hello World!\n");
}
```

Each quick-start procedure gives only basic instructions for compiling, linking, and running a C program. See the following chapters for detailed information:

TSO Quick-Start

Use this procedure to compile, link, and run a simple C program from the TSO environment.

1 Write a small "Hello World!" program and store it in *userid*.QSTART.C(HELLO).

Note: The transient run-time library must be allocated to the DDname CTRANS or installed in the system link list before you can use the SAS/C Compiler. Your installation will probably cause it to be allocated automatically. If you encounter difficulty with the following steps, use the TSO ALLOCATE command to associate the library with the DDname CTRANS. △

2 Enter the following command to compile the C source code stored in *userid*.QSTART.C(HELLO):

```
LC370 QSTART(HELLO)
```

The object code output is stored in *userid*.QSTART.OBJ(HELLO).

3 Enter the following command to link the HELLO program:

```
COOL QSTART(HELLO)
```

The load module is stored in *userid*.QSTART.LOAD(HELLO).

4 Enter the following command to run the HELLO program:

```
CALL QSTART(HELLO)
```

The program executes and "Hello world!" is displayed.

CMS Quick-Start

Use this procedure to compile, link, and run a simple C program from the CMS environment.

1 Write a small "Hello world!" program and store it in a file named HELLO C.

2 Enter the following command to compile the C source code stored in HELLO C:

```
LC370 HELLO
```

The object code output is stored in HELLO TEXT.

3 Enter the following command to link the HELLO program:

```
COOL HELLO (GENMOD HELLO
```

The load module is stored in HELLO MODULE.

4 Enter the following command to run the HELLO program:

```
HELLO
```

The program executes and "Hello world!" is displayed.

USS Quick-Start

Use this procedure to compile, link, and run a simple C program from the USS shell.

Note: The transient library must be defined before a SAS/C program can be executed under the shell unless the library modules have been installed in the system link list. If your site does not define this library automatically, you will need to assign the transient library data set name to the environment variable ddn_CTRANS and export it before running the compiler. △

1. Write a small "Hello world!" program and store it in a hierarchical file system (HFS) file named **`hello.c`**.

   Note: In order to invoke the **`sascc370`** command, you must include the directory where SAS/C was installed in your PATH environment variable. Probably, your site will define PATH appropriately for you when you start up the shell. If your site does not do this, contact your SAS Software Representative for C compiler products to obtain the correct directory name and add it to your PATH. △

2. Enter the following command to compile and link the C source code stored in **`hello.c`**:

   ```
   sascc370 -o hello hello.c
   ```

   The object code output is stored in **`hello.o`**, and the executable output is stored in **`hello`**.

3. Enter the following command to run the **`hello`** program:

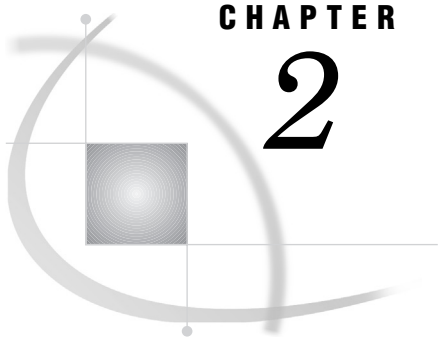
   ```
   hello
   ```

   The program executes and "Hello world!" is displayed.

Summary of Changes and Enhancements

For details on changes and enhancements to the SAS/C and C++ compiler, see the following documents:

- *SAS/C Software: Changes and Enhancements, Release 7.00*
- *SAS/C Software: Changes and Enhancements, Release 6.50*



CHAPTER

2

Source Code Conventions

Introduction

The C language accepted by Release 6.00 of the SAS/C Compiler corresponds to that specified by the ISO/ANSI C Standard (American National Standards Committee 1989). This chapter discusses in more detail the C language accepted by the compiler.

"Environmental Elements" on page 11 below and "Language Elements" on page 24 provide specific SAS/C information about environment-dependent aspects of the SAS/C implementation. This discussion is structured like the ISO/ANSI Standard. Comparisons are made to relevant sections of Appendix A, "C Reference Manual," in The C Programming Language, Second Edition (Kernighan and Ritchie 1988).

"Language Extensions" on page 28 describes SAS/C extensions to the C language as described by the ISO/ANSI Standard. While these extensions can be very useful on IBM mainframes, they are nonportable.

"Implementation-defined Behavior" on page 38 documents the behavior of the SAS/C Compiler and Library in areas the ISO/ANSI Standard leaves open for definition by a particular implementation of the C language.

Environmental Elements

This section covers the following environmental elements, which affect the SAS/C Compiler's processing of source code:

- Source File Sequence Number Handling
- Include-File Processing
- Multibyte Character Support
- Special Character Support
- Escape Sequences
- Translation Limits
- Storage Class Limits
- Numerical Limits

Source File Sequence Number Handling

The compiler examines the first record in the source file and in each **`#include`** file to determine if that file contains sequence numbers. Therefore, you can safely mix files with and without sequence numbers and use the compiler on sequenced or nonsequenced files without worrying about specifying a sequence number parameter.

For a file with varying-length records, if the first four characters in the first record are alphanumeric and the following four characters are numeric, then the file is assumed to have sequence numbers.

For a file with fixed-length records, if the last four characters in the first record are all numeric and the preceding four characters are alphanumeric, then the file is assumed to have sequence numbers.

If a file is assumed to have sequence numbers, then the characters in each record at the sequence number position are ignored. The characters are not treated as part of the program by the compiler, and they are printed in the sequence number position on the listing, rather than where they actually appear in the record.

This algorithm detects sequence numbers or their absence correctly for almost all files, regardless of record type or record length. Occasionally it may cause problems, as in the following examples:

- For a file in which only some records, not including the first record, contain sequence numbers, the validity of the sequence number is questionable. The entire record is treated as C code, so errors are certainly generated.
- Another problem is a file of fixed-length records in which the last eight characters of the first record resemble a sequence number but are instead, for example, a long numeric constant. A dummy first record fixes this problem.

Include-File Processing

Because of the variety of environments in which the SAS/C Compiler executes, and the variety of include-file organizations used by programs of different sorts, the

compiler's include-file processing can be quite complex. The implementation was designed to meet the following requirements:

1. Include files can be stored in any file type readable by the compiler, including OS/390 PDSs (partitioned data sets), CMS minidisks or shared file system files, and UNIX System Services (USS) HFS files.
2. The user can easily specify the locations where include files reside, in any environment.
3. The user can choose between UNIX oriented or mainframe oriented search rules.
4. Provision is made to allow include statements specifying a directory path to be resolved from a file system that does not support directories by means of a header name mapping facility.

The rest of this section presents include-file processing topics in the following order.

1. Processing in simple OS/390, CMS, and USS applications, in which no **include** directives specify directory paths.
2. Processing in applications that specify directory paths in **include** directives, but where the include files are stored in a mainframe file system (for example, a CMS minidisk).
3. Processing in applications that access include files from the USS hierarchical file system, or that require more flexibility than provided by the facilities described in the first two sections.

Simple include-file processing under OS/390

On OS/390, the accepted forms for a simple **#include** directive are as follows:

```
#include "member.libddn"
#include "style:pathname"
#include "//[style:]pathname"
#include <member.h>
#include <member>
```

In the first form, the first part of the filename is interpreted as a member name and the second part as a DDname. For instance, an include of **"protos.h"** will look in DDname H for a PDS member PROTOS. Note that the DDname H could also reference an USS HFS directory, in which case the compiler includes the file **protos.h** from that directory. If the member specified is not found in the DDname specified, the compiler processes the directive as if the filename were enclosed in angle brackets. In the example, if the compiler failed to locate **"protos.h"**, it would attempt to include **<protos.h>**.

In the second or third form, the filename must be a valid pathname for a call to **fopen** (see SAS/C Library Reference, Volume 1). In this case, the compiler simply includes the file specified. If the file cannot be opened, the compilation fails.

In the fourth and fifth forms, the first part of the filename is interpreted as a member name, and the rest of the filename, if any, is ignored. The include file is read from the DDname SYSLIB. If the member cannot be found, the compilation fails.

Note that if the underscore character (_) appears in the member or libddn part of an include filename, it is translated to a pound sign (#) wherever it occurs, except when including from an HFS directory. For instance, including **"name_1.my_lib"** would fetch the member NAME#1 from the PDS referenced by the DDname MY#LIB. Also note that if the member name or libddn name is longer than eight characters, it is truncated.

Simple include-file processing under CMS

On CMS, the accepted forms for a simple **#include** directive are as follows:

```
#include "filename.filetype"
#include "style:pathname"
#include "//[style:]pathname"
#include <member.h>
#include <member>
```

In the first form, the filename is interpreted as a CMS filename and filetype and located on an ACCESSed minidisk. For instance, an include of **"protos.h"** will search all ACCESSed minidisks for the file PROTOS H. If the file specified cannot be found, the compiler processes the directive as if the filename were enclosed in angle brackets. In the example, if the compiler failed to locate **"protos.h"**, it would attempt to include **<protos.h>**.

In the second and third forms, the filename must be a valid pathname for a call to **fopen** (see SAS/C Library Reference, Volume 1). In this case, the compiler simply includes the file specified. If the file cannot be opened, the compilation fails.

In the fourth and fifth forms, the first part of the filename is interpreted as a member name, and the rest of the filename, if any, is ignored. The include file is read from the currently GLOBALed MACLIBs. If the member cannot be found, the compilation fails.

Note that if a member name or filename component is longer than eight characters, it is truncated.

Simple include-file processing under the USS shell

Under the USS shell, the accepted forms for a simple **#include** directive are as follows:

```
#include "pathname"
#include "//[style:]pathname"
#include <member.h>
```

In the first form, the filename is interpreted as an HFS filename in the same directory as the including file. For instance, an include of **"protos.h"** in the source program will search the source directory for the file **protos.h**. If the file specified cannot be found, the compiler processes the directive as if the file were enclosed in angle brackets. In the example, if the compiler failed to locate **"protos.h"**, it would attempt to include **<protos.h>**.

In the second form, the filename must be a valid pathname for a call to **fopen** (see SAS/C Library Reference, Volume 1). In this case, the compiler simply includes the file specified. If the file cannot be opened, the compilation fails.

In the third form, the first part of the filename is interpreted as a member name, and the rest of the filename, if any, is ignored. The member is read from the site-defined OS/390 PDS containing the standard header files. If the member cannot be found, the compilation fails.

Note that complex applications may require the use of the **ipath** compiler option to augment these simple search rules. This option is discussed in "Complete include processing" on page 16.

Header file mapping

Many existing programs written for other environments such as UNIX contain **#include** directives which reference directories or which have long filenames that are not unique within the first eight characters. Since the OS/390 and CMS file systems do not support directories or filename components longer than eight characters, these files could not be stored on the mainframe without being renamed and/or reorganized. Header file mapping was developed for the SAS/C Compiler to allow the portable source for these applications to be used despite the deficiencies of the mainframe file systems.

SAS/C allows a user to define a lookup table to the compiler, which defines a mapping from include filenames as specified in the program to actual filenames. Because the table is external to the program, the program's source need not be changed, and the same table can be used for multiple programs. There are two header mapping files: one for system (angle-bracketed) include files and one for user (double-quoted) include files.

Locating the header map

Before opening any include file, the compiler attempts to open a $$HDRMAP file. If the include file is a system header file, the header map file is found like **<$$hdrmap.h>** (that is, on OS/390 it is the $$HDRMAP member of SYSLIB, and on CMS it is the $$HDRMAP member of the GLOBALed MACLIBs). Similarly, if the include file is a user include file, the compiler looks for $$HDRMAP as if it were included as **"$$hdrmap.h"**. Note that on OS/390 the libddn part of the original user include filename is ignored, and $$HDRMAP is always fetched from the DDname H.

Note that $$HDRMAP is processed for any **include** directive, even if the filename does not include a directory specification.

Format of the header map

The $$HDRMAP file consists of one or more mapping lines and comments. A comment is any line whose first non-white-space character is a pound sign (#). A mapping line is a line containing two strings separated by blanks or tabs and optionally followed by comments. For example:

```
X11/AtomMgr.h        XATMGR.H       X11R5 header
```

The first name is the expected header filename. The filename from a compiled **#include** directive is compared with the header name in each $$HDRMAP line until a matching line is found. The names must match exactly. For example, **"./X11/AtomMgr.h"** and **"X11/AtomMgr.h"** are not considered to match. If the compiled filename does match the first name of a header map line, the compiler replaces it with the second string on the line and proceeds to search for that file using the rules described in "Include-File Processing" on page 11. Thus, on OS/390, a **#include** directive for **"X11/AtomMgr.h"** would attempt to include the member XATMGR in the DDname H.

Note that a target filename in the $$HDRMAP file should not be bracketed or quoted. The type of file (system or user) is always assumed to be the same as in the original **#include** directive. Also note that if the search for a user header file (whether the name is mapped or not) fails, the system $$HDRMAP file is used to map the name before the system header file location is searched. Therefore, the order of steps for a user header file is as follows:

1. Look for a matching entry in the user $$HDRMAP. If found, replace the original name by the mapped name.
2. Search the user header file location for the filename resulting from step 1.
3. If that fails, look for an entry matching the original name in the system $$HDRMAP. If found, replace the original name by the mapped name.
4. Search the system header file location for the filename resulting from step 3.

Header map example

Assume the following system header file map (SYSLIB($$HDRMAP) on OS/390):

```
X11/AtomMgr.h                XATMGR.H
X11/Xlibos.h                 XLIBOS.H
```

```
../mydir/george.h            GEORGE.MYDIR
```

and the following user header file map (H($$HDRMAP) on OS/390):

```
MyVeryLongName.h             MVLN.H
../mydir/myfile.h            MYFILE.MYDIR
```

Then, the following **#include** directives would be processed as follows:

#include "../mydir/myfile.h"
The second line of the user header map would be located, and the compiler would attempt to read the include file from member MYFILE of the DDname MYDIR.

#include "../mydir/george.h"
No match is found in the user header file map. Assuming that there is no member GEORGE in the file referenced by DDname H, the system map is then searched and the third line is a match. Because the match was found in the system map, GEORGE.MYDIR is treated as a system header filename, and the compiler attempts to read member GEORGE of DDname SYSLIB.

#include <X11/Xlibos.h>
The compiler searches the system header file map and finds the second line. The compiler reads member XLIBOS of DDname SYSLIB.

#chain Command

#chain *FILENAME*

When the **#chain** command is specified in a $$HDRMAP file, the header-map lookup uses the **#include** processing rules specified on this compilation to locate the file named FILENAME. It uses system include rules when processing system $$HDRMAP files and user include rules for user defined $$HDRMAP files.

Any header-map entries found in the file will be inserted in the list at the point of the **#chain** command.

A file that is processed using a **#chain** can **#chain** other files. If the file named on the **#chain** cannot be located or opened using the include search rules, the **#chain** is ignored and processing continues.

For example, in a user include $$hdrmap.h, you could have

```
foo.h bar.h
harry.h george.h
#
# Go get the mappings for the current project.
#

#chain //DDN:PROJECT(MAPPINGS)

alice.h betty.h
```

If the DDN “PROJECT” was defined so that MAPPINGS contained

```
somebiglongname.h sbln.h
```

Then the header map list would be :

```
foo.h ---> bar.h
harry.h ---> george.h
somebiglongname.h ---> sbln.h
alice.h ---> betty.h
```

Complete include processing

This section describes in detail the entirety of the compiler's header file processing. This section is most relevant if you have header files in the USS hierarchical file system, but it may also be relevant to other complicated applications.

To deal with the complexities introduced by HFS directories and to provide a high level of compatibility with UNIX include-file processing, the compiler offers the **ipath** and **usearch** options. Additionally, on CMS the _HEADERS environment variable provides additional flexibility for users of the CMS shared file system. Refer to "Using Environment Variables to Specify Defaults" on page 85 for information on the _HEADERS environment variable.

The **usearch** option specifies "UNIX search rules," which is a rearrangement of the compiler's normal methods of processing for greater UNIX compatibility. **usearch** is the default when you compile under the USS shell, but it may be specified for compiles from other environments. The **ipath** option allows you to specify one or more locations to be searched for header files. These are normally HFS directories, but they could also be PDS names or CMS shared file system directories.

If the **usearch** option is not in effect, the order of include processing for a **#include** of a user (double-quoted) header file is as follows. (Note that if any step locates the file, the remaining steps are not performed.)

1. A $$HDRMAP lookup is performed to see if the filename should be mapped to another name.
2. If the filename was specified with a style prefix, the compiler attempts to open the file specified. Similarly, if the compiler is running from the USS shell and the include file is specified using an absolute pathname (one beginning with a slash), the compiler attempts to open the file specified. In either case, if this fails, no further steps are attempted.
3. The locations specified by **ipath** options are searched for the header file, in the order in which they were specified.
4. If the file containing the **include** directive was obtained from an HFS directory, the compiler searches this directory.
5. If the compiler is running under the USS shell, it searches the current directory.
6. The compiler looks in the normal, system-dependent place for the header file. On OS/390, it transforms the filename into an OS/390 member name and DDname, as described in "Simple include-file processing under OS/390" on page 12, discarding any directory specification. On CMS, this search takes place in two stages:
   a. If the _HEADERS environment variable was defined, as described in "Specifying Shared File System Directories" on page 86, the compiler searches each shared file system directory specified by _HEADERS for the required file.
   b. Any directory specification in the filename is discarded, and all ACCESSed minidisks are searched for the file.
7. The compiler attempts to locate the file as a system header file.

If the **usearch** option is not in effect, the order of processing for a system (bracketed) include file is as follows. (Note that if any step locates the file, the remaining steps are not performed.)

1. A $$HDRMAP lookup is performed to see if the filename should be mapped to another name.
2. If the filename was specified with a style prefix, or if the compiler is running under the shell and an absolute pathname was specified, the compiler attempts to open the file specified. If that fails, no further steps are attempted.
3. The compiler searches each location specified via the INCLUDE environment variable. (This facility is defined for use in library development and is not recommended as a customer interface.)

4 The compiler looks for the header file as a member of SYSLIB on OS/390 or of the GLOBAL MACLIBs on CMS.

If the **usearch** option is specified to enforce UNIX search rules, the search for a user (double-quoted) include file proceeds as follows. (Note that if any step locates the file, the remaining steps are not performed.)

1 A $$HDRMAP lookup is performed to see if the filename should be mapped to another name.
2 If the filename was specified with a style prefix, or if the compiler is running under the shell and an absolute pathname was specified, the compiler attempts to open the file specified. If that fails, no further steps are attempted.
3 The compiler looks for the header file in the location (HFS directory, OS/390 PDS, CMS minidisk or shared file system directory) containing the file that included this one.
4 The locations specified by **ipath** options are searched for the header file, in the order in which they were specified.
5 The compiler looks in the normal, system-dependent place for the header file. On OS/390, it transforms the filename into an OS/390 member name and DDname, as described in "Simple include-file processing under OS/390" on page 12, discarding any directory specification. On CMS, this search takes place in two stages:
   a If the _HEADERS environment variable was defined, as described in "Specifying Shared File System Directories" on page 86, the compiler searches each shared file system directory specified by _HEADERS for the required file.
   b Any directory specification in the filename is discarded, and all ACCESSed minidisks are searched for the file.
6 The compiler attempts to locate the file as a system header file.

If the **usearch** option is specified to enforce UNIX search rules, the search for a system (bracketed) include file proceeds as follows. (Note that if any step locates the file, the remaining steps are not performed.)

1 A $$HDRMAP lookup is performed to see if the filename should be mapped to another name.
2 If the filename was specified with a style prefix, or if the compiler is running under the shell and an absolute pathname was specified, the compiler attempts to open the file specified. If that fails, no further steps are attempted.
3 The locations specified by **ipath** options are searched for the header file, in the order in which they were specified.
4 The compiler searches each location specified via the INCLUDE environment variable. (This facility is defined for use in library development and is not recommended as a customer interface.)
5 The compiler looks for the header file as a member of SYSLIB on OS/390 or of the GLOBAL MACLIBs on CMS.

The primary differences between the UNIX (**usearch**) search rules and the ordinary SAS/C (**nousearch**) search rules are the following:

- When **usearch** is in effect, the locations specified by **ipath** are searched for both user and system header files. When **usearch** is not in effect, **ipath** is used only to search for user header files.
- When **usearch** is in effect, the including file's directory is searched before the **ipath** directories. When **usearch** is not in effect, the **ipath** directories are searched before the directory of the including file.
- When **usearch** is in effect, the location of the including file is always searched. When **usearch** is not in effect, it is searched only if the including file was obtained from the HFS.

- When **usearch** is not in effect and the compiler is running under the shell, an explicit search of the current directory is performed before the system-dependent search. This step is not performed when **usearch** is in effect.

Multibyte Character Support

Multibyte character support enables programs to adapt to different cultures by providing support for large character sets. (See Chapter 10, "Localization," and Chapter 11, "Multibyte Character Functions," in SAS/C Library Reference, Volume 2 for more information.) To support large character sets (for example, the 14,000 or so most commonly used Japanese ideographs), at least 2 bytes are needed to provide encoding for the complete set. An 8-bit byte alone cannot provide the number of distinct values necessary to provide this support.

The ISO/ANSI Standard defines the concepts of multibyte characters and wide characters to support these large character sets. The method of implementation of these concepts is implementation-defined. The compiler has chosen to implement these features compatibly with the EBCDIC DBCS (double-byte character set) definition when the current locale at compile time supports DBCS.

A wide character is a value of type **wchar_t**, which the compiler defines as **unsigned short**. If the value is in the range of the **char** type, it represents a standard EBCDIC character. If the value is outside the range of **char**, it represents an extended character, such as a character of the Japanese Kanji character set. EBCDIC DBCS constrains each byte of a wide character outside the range of **char** to have a value between 0x40 and 0xFE.

Wide characters are easy to process in C, but external DBCS data are usually not in this format. The multibyte character string format is more common. A multibyte character string contains a mixture of standard EBCDIC single-byte characters and extended double-byte characters. (This is called a mixed string in EBCDIC DBCS terminology.)

When DBCS support is enabled, the interpretation of the bytes of a multibyte character string is controlled by use of the SO (shift out) and SI (shift in) characters whose values are 0x0E and 0x0F, respectively. At the start of a multibyte string, characters are interpreted as single-byte EBCDIC characters until an SO character is encountered. This causes a transition to the double-byte shift state in which each pair of characters is interpreted as an extended double-byte character. If an SI character is encountered, the single-byte interpretation is resumed, and so on. Thus, the interpretation of individual characters in a multibyte character string is dependent on the current shift state.

When DBCS support is not enabled, all characters are interpreted as standard EBCDIC characters, and no special semantics are associated with SO or SI.

Note that the term multibyte character is used to refer to any character of the extended character set. In particular, standard EBCDIC characters such as A are multibyte characters even though only a single byte is required to represent them.

The following two sections cover aspects of the multibyte character implementation related to compiler support. Chapter 11 in SAS/C Library Reference, Volume 2 covers the library implementation of the multibyte functions.

Control of character types by locale support

The compiler uses the current locale to enable its DBCS support. This is controlled by the environment variable _LOCALE as described in Chapter 10 in SAS/C Library Reference, Volume 2. For example, the locale that needs to be set is the one in use when running the compiler (not when executing the resultant program).

The compiler uses the locale in effect during compilation to enable its DBCS support. This is controlled by the environment variable _LOCALE. Under OS/390 batch, the environment variable is set by including =_LOCALE=DBCS in the compiler's PARM string. In TSO, you can use the PUTENV command to set _LOCALE, if your site makes this command available. Under CMS, assigning the value DBCS to the GLOBALV variable _LOCALE in the CENV group has the same effect. Note that, because GLOBALV variables are shared by all programs, this assignment also affects any C program that uses the default locale during execution. To limit the effect to just the compiler under CMS, you should use an explicit command-line assignment, as you would under OS/390.

Compiler lexical processing

The values assigned to characters are implementation-defined. For all basic characters and other single-byte characters, the values are those given by EBCDIC. For multibyte characters, the values given by the DBCS for the language character set currently enabled are used.

All instances of DBCS in C source are in mixed strings. For string literals (including array initializers) and character constants, an L prefix controls whether or not the type of the literal or constant is based on **`char`** or **`wchar_t`**.

The ISO/ANSI C Standard allows the following lexical elements to contain multibyte characters (or any other members of the extended character set):

- character constants
- string literals
- header filenames
- array initializers.

Character constants

There are two kinds of character constants: integral and wide. Wide character constants are prefixed by the letter L. Integral character constants have type **`int`**; wide character constants have type **`wchar_t`**. In the SAS/C implementation, **`wchar_t`** is an **`unsigned short`**.

An integral character constant normally contains only a single character. If more than one character is present, the value of the constant is the value of the integer whose rightmost bytes are those of the constant. (If there are more than four characters, only the four rightmost are used.) For instance, the integral character constant 'HI' has the same value as 0xC8C9. Note that use of an integral character constant containing more than one character is not portable.

A wide character constant normally contains only a single multibyte character or escape sequence. If a wide character constant contains more than one, only the rightmost is used. The value of a wide character constant is determined as follows:

- If it contains a single-byte character or an escape sequence other than an octal or hexadecimal escape sequence, the value of the wide character constant is the same as that of the corresponding integral character constant.
- If it contains a double-byte character, the value of the wide character constant is the value of the two bytes between the SO and SI characters that delimit the double-byte character in the source file.
- If it contains a hex or octal escape sequence, the sequence is interpreted as a **`wchar_t`** value, using its rightmost bits; any excess bits are ignored.

String literals

The implementation of string literals and wide string literals is nearly identical to that of character constants. The type of a wide string literal is interpreted as an array of **wchar_t**. The compiler issues a warning message for attempts to use adjacent string concatenation with different types of string literals and gives the type for the first literal in the concatenation. (The ISO/ANSI Standard treats this as undefined.)

Header filenames

The compiler accepts multibyte sequences in header filenames. The mixed DBCS string is passed to the operating system unchanged.

Array initializers

A wide string literal may be used to initialize an array of elements whose type is **wchar_t**. Both static and auto array initialization are supported.

Special Character Support

The C language uses a number of special characters. Many IBM mainframe terminals and printers do not supply all of these characters. The compiler provides three solutions to this problem:

- ISO/ANSI trigraphs
- digraphs
- a special character translation table.

Trigraphs sequences

Trigraph sequences are an invention of the C standardization committee. They are intended to serve as replacements for characters in the C character set that do not appear in the ISO 646 character set. See the **trigraphs** option in Chapter 6, "Compiler Options," on page 101 for information about trigraphs.

Digraphs

The C language is implemented traditionally using the ASCII character set. The compiler uses EBCDIC, the IBM 370 preferred character set, as its native set. Because some characters used by the C language are not normal EBCDIC characters (that is, they do not appear on many terminal keyboards), alternate representations are available for them. Also, for some characters there is more than one similar EBCDIC character. In such cases, the compiler accepts either.

Table 2.1 on page 20 gives alternate representations that the compiler accepts.

Table 2.1 Digraph Sequences For Special Characters

C Character	EBCDIC Values(s) (hex)	Alternate Forms
[(left bracket)	0xad	(|
] (right bracket)	0xbd	|)
{ (left brace)	0x8b, 0xc0	\(or (<

C Character	EBCDIC Values(s) (hex)	Alternate Forms
} (right brace)	0x9b, 0xd0	\) or >)
| (inclusive or)	0x4f, 0x6a	\!
~ (tilde)	0xa1	¬
\ (backslash)	0xe0, 0xbe	(see discussion below)

For all symbols except the backslash (), substitute sequences are not replaced in string constants or character constants. For example, the string constant "(|" contains two characters, not a single left bracket. (Contrast this behavior with the standard trigraphs, which are replaced in string and character constants.)

The backslash is a special case because it has significance within string and character constants as well as within C statements. However, the compiler can be customized to accept an alternate single character for the backslash. In addition, the customization can include alternate single character representations of the characters in Table 2.1 on page 20. The default alternate representations are listed in Table 2.2 on page 22. See your SAS Software Representative for C compiler products for more information.

Note that in addition to the digraphs described in Table 2.1 on page 20, your site can customize the compiler to accept alternate single-character representations of the characters. See your site representative for details.

Special character translate table

The special character translate table enables each site to customize the representations of special characters. That is, the site decides which hexadecimal bit pattern or patterns represent that character and, thus, can choose a representation that is available on their terminals and printers. The special character translate table enables you to choose a representation of all the unusual C characters, in addition to the exclamation point (!).

The special characters that can be customized are braces ({}), square brackets ([]), not sign (¬), tilde (~), backslash (), the vertical bar (|), the exclamation point (!), and the pound sign (#). You should determine if your site has customized values for these characters and determine values. Otherwise, the default representations listed in Table 2.2 on page 22 are in effect. Consult your SAS Software Representative for C compiler products for details of customized values.

Table 2.2 on page 22 shows the four possible default representations for each character. The compiler and OMD (object module disassembler) accept either of two representations of the character in a source file. These are the primary and alternate representations in columns two and three. The compiler and OMD can produce either of two representations of each character in a listing file depending on the execution options specified. The first is the standard print representation shown in column four. The second, the overstrike representation listed in column five, is produced by overstriking the character with another character. (Details about the execution options and how they interact are provided in Chapter 6, “Compiler Options,” on page 101.) The entries in columns two through five are EBCDIC equivalents of the characters in hexadecimal notation.

Keep in mind that these alternate representations for characters apply only to C program source and not to the contents of files read by C programs.

In addition to the problem of entering C source text on devices that do not have the full character set, there is also the problem of printing the text. The compiler supports two options to assist in this area: the **`trans`** option and the **`upper`** option. See Chapter 6, “Compiler Options,” on page 101 for more information on these options.

Note: Do not use an underscore (_) as an overstrike character. When you specify an alternate character in the source file, the compiler places an underscore under the alternate character in the listing file. If you use an underscore as an overstrike character, it will not be clear whether the underscore in the listing file represents an alternate character or an overstrike character. △

Table 2.2 Default Representations for Special Characters

	Source File Representation		Listing File Representation	
Character	Primary	Alternate	Print	Overstrike
	0xc0	0x8b	0x8b	0x4c / 0x4f
left brace	{	{	{	< with | overstrike
right brace	0xd0	0x9b	0x9b	0x6e \ 0x4f
	}	}	}	> with | overstrike
left bracket	0xad	0xad	0xad	0x4c / 0x60
	[	[	[	< with - overstrike
right bracket	0xbd	0xbd	0xbd	0x6e / 0x60
				> with - overstrike
not sign (exclusive or)	0x5f	0x71	0x5f	0x5f (no overstrike)
tilde	0xa1	0xa1	0xa1	0x7d / 0x60
	~	~	(degree symbol)	' with - overstrike
backslash	0xe0	0xbe	0xbe	0x7e
vertical bar | or (inclusive or)	0x4f	0x6a	0x4f	0x4f (no overstrike)
	|		|	|
pound sign	0x7b	0x7b	0x7b	0x7b (no overstrike)
	#	#	#	#
exclamation point	0x5a	0x5a	0x5a	0x5a (no overstrike)
	!	!	!	!

Escape Sequences

The compiler produces a unique **`char`** value for certain alphabetic escape sequences that represent nongraphic characters. The characters' associated hex values and ISO/ANSI meaning appear in Table 2.3 on page 23.

Table 2.3 Escape Sequence Values

Sequence	Hex Value	Meaning
\a	0x2f	alert
\b	0x16	backspace
\f	0x0c	form feed
\n	0x15	new line
\r	0x0d	carriage return
\t	0x05	horizontal tab
\v	0x0b	vertical tab

Translation Limits

Wherever possible, the compiler avoids imposing fixed translation limits. The ISO/ANSI Standard translation limits for which the compiler does impose a fixed limit are listed in Table 2.4 on page 24, followed by some other limits of practical interest. If no limit is listed, the only limit is imposed by the memory available to the compiler or by the program when it is executed. The **extname** option can override the following translation limits for internal and external identifiers and allow 64K characters of significance:

- Internal identifiers have 31 characters of significance.
- External identifiers have 8 (monocase) characters of significance.
- Forty arguments are the maximum allowed in a macro definition.
- Individual objects can be up to 8 megabytes in size. The compiler imposes no limit on array sizes. Overall limits for each storage class are listed in "Storage Class Limits" on page 23.
- The maximum level of **#include** file nesting is 16.

Storage Class Limits

The total size of all objects declared in one compilation with the same storage class is limited according to the particular class, as follows:

- extern - 16777215 (16M-1) bytes
- static - 8388607 (8M-1) bytes
- auto - 8388607 (8M-1) bytes
- formal - 65535 (64K-1) bytes.

Note that the following types of programs generate very large CSECTS:

- programs compiled with the **norent** option and with large amounts of static or defined external data or both
- programs compiled with the **rentext** option and with large amounts of static data.

You should consider alternatives to using large amounts of static data (for example, dynamic storage allocation via **malloc**).

Storage allocated via the **malloc** family of routines is limited only by available memory.

Numerical Limits

The numerical limits are what you would expect for a 32-bit, two's complement machine such as the IBM 370. Table 2.4 on page 24 shows the size ranges for the integral types.

Table 2.4 Integral Type Sizes

Type	Length in Bytes	Range
char	1	0 to 255 (EBCDIC character set)
signed char	1	-128 to 127
short	2	-32768 to 32767
unsigned short	2	0 to 65535
int	4	-21474883648 to 2147483647
unsigned int	4	0 to 4294967295
long	4	-2147483648 to 2147483647
unsigned long	4	0 to 4294967295

Table 2.5 on page 24 shows the size ranges for **`float`** and **`double`** types.

Table 2.5 Float and Double Type Sizes

Type	Length in Bytes	Range
float	4	+/-5.4E-79 to +/-7.2E75
double	8	+/-5.4E-79 to +/-7.2E75
long double	8	+/-5.4E-79 to +/-7.2E75

For more details on the implementation of the various data types, see Chapter 3, "Code Generation Conventions," on page 45.

Language Elements

This section describes standard C constructs for which there are special SAS/C considerations.

Constants

String literals

By default, identically written string constants refer to the same storage location; that is, only one copy of the string is generated by the compiler. The **`nostringdup`**

compiler option can be used to force a separate copy to be generated for each use of a string literal. However, modifying string constants is not recommended and renders a program non-reentrant.

Strings that are used to initialize **char** arrays (not **char ***) are not actually generated because they are shorthand for a comma-separated list of single-character constants.

Conversions

Pointer conversion

Implicit pointer conversion by assignment is allowed but generates a warning message (which is a diagnostic in the sense of the C Standard) whenever any value other than a pointer of the same type, a pointer to **void**, the constant 0, or **NULL** is assigned to a pointer. There is an important reason for the warning. Although many C programs make the implicit assumption that pointers of all types can be stored in **int** variables or other pointer types and retrieved without difficulty (and this is true for the SAS/C Compiler), the language itself does not guarantee this. On word-addressed machines, for example, such conversions do not always work properly. The warning message provides a gentle (and nonfatal) reminder of this fact. A cast operator can be used to eliminate the warning. The cast then indicates that the conversion is intentional and not the result of improper coding.

A more stringent requirement is enforced for initializers, where the expression to initialize a pointer must evaluate to a pointer, **null**, or an integral constant 0. Any other value is an error.

When pointers of different types are compared, the operand on the right side of the comparison is converted to the type of the operand on the left side of the comparison; comparison of a pointer and one of the integral types causes a conversion of the integer to the pointer type. Both of these operations are of questionable value unless pointers are being compared for equality or inequality. Note that the result of a relational pointer comparison is undefined (according to ISO/ANSI) if the pointers do not address elements of the same array, structure, or union object.

Declarations

Function prototypes

The standard header files contain prototypes for library functions as specified by the C Standard. Should you want to avoid the use of these prototypes, the preprocessor variable **_NOLIBCK** enables you to do this. To bypass the prototypes, precede the **#include** statement for a header file with a **#define** statement in the following form:

```
#define _NOLIBCK
```

You can also define **_NOLIBCK** from the command line using the **define** compiler option. Use of **_NOLIBCK** to suppress library prototypes is strongly discouraged.

Structure and union type names

The compiler accepts unions that have no identifier. See “Anonymous unions” on page 33 for more information.

Bitfields

The compiler optionally allows bitfields to have a type other than **int**, **signed int**, or **unsigned int**. See "Noninteger bitfields" on page 34 for more information.

Predefined Macro Names

The compiler provides the following predefined macro names required by the C Standard:

__DATE__
__FILE__
__LINE__
__STDC__
__TIME__

These macros are useful for generating diagnostic messages and inline program documentation.

__DATE__ specifies the current date, in the form "Mmm dd yyyy" (for example, Jan 01 1991).

__FILE__ expands to a string constant containing the current filename. The exact form of the name is system-dependent and also depends on the type of the file. In general, the name of the **__FILE__** will be a canonicalized version of the source or include filename. If the file is an USS HFS file, **__FILE__** contains an absolute pathname, for example, **"//HFS:/u/howard/hdrs/protos.h"**.

__LINE__ expands to an integer constant that is the relative number of the current source line within the file (primary source file or **#include** file) that contains it.

__STDC__ specifies the decimal constant 1.

__TIME__ specifies the current time, in the form "hh:mm:ss".

None of the above predefined macros can be undefined with **#undef**.

The compiler provides the following predefined macro names in addition to the names specified by the ISO/ANSI Standard. The automatic definition of these names can be collectively suppressed by using the **undef** compiler option. See Chapter 6, "Compiler Options," on page 101 for more information.

```
#define DEBUG    1  /* if DEBUG option is used             */
   #define NDEBUG   1  /* if DEBUG option is not used         */
   #define OSVS     1  /* if compiling under TSO or MVS batch */
   #define CMS      1  /* if compiling under CMS              */
   #define __I370__    1  /* indicates the SAS/C Compiler        */
```

The **#undef** preprocessor directive can undefine individually these macro names.

Some predefined macro names indicate certain compiler options. See "Preprocessor Options Processing" on page 128 for more information.

The compiler creates two other preprocessor symbols in addition to the option symbols:

__SASC__ is assigned the current release number of SAS/C software. For example, the preprocessor symbol assignment for Release 6.00 is equivalent to the following definition:

```
#define __SASC__ 600
```

__COMPILER__ is assigned the current release of SAS/C software as a string. For Release 6.00, this is equivalent to the following definition:

```
#define __COMPILER__ "SAS/C Version 6.00x "
```

where *x* is the product release letter.

Data Types

Data types are sets of values combined with sets of allowable operations that apply to each member of a specific data type. The members of a data type can assume only those values and return only those functions contained in that data type.

long long data type Support

SAS/C Release 7.00 supports the **long long** data type, both signed and unsigned. This data type is used for 64-bit (8-byte) integers. To specify either type in a declaration or cast, use the keyword **long** twice, and the **unsigned** keyword when necessary, as shown in the following example:

```
long long taxes();
if (taxes() >= (unsigned long long) my_salary)
    panic();
```

Description of the long long Data Type

A **signed long long** can hold any integer value between -9223372036854775808 (-2 to the 63rd power) to 9223372036854775807 (2 to the 63rd power - 1). These values are associated with the symbols **LLONG_MIN** and **LLONG_MAX** (defined in **limits.h**), respectively. An **unsigned long long** can hold any value between 0 and 18446744073709551615 (2 to the 64th power - 1). This value is associated with the symbol **ULLONG_MAX** defined in **limits.h**.

Constants in the range of the **long long** types may be written in decimal, octal, or hexadecimal form, the same as other integer constants. If a constant is too large to be an unsigned long, it is assumed to be a **long long** or **unsigned long long** constant. **long long** constants in the range of a 32-bit integer type can be written using the suffix LL (for **signed long long**) or ULL (for **unsigned long long**). Examples of **long long** constants are 14LL and 0XFF00FF00ULL.

long long values can be converted to and from all other numeric types, sometimes with a loss of accuracy. Note that a **long long**, with 63 bits of accuracy, is more accurate than a **double**, with 53-56 bits, and that therefore conversions from **long long** to **double** may well lose precision. Note that if an expression has a **double** (or **float**) operand and a **long long** operand, the **long long** operand is converted to a **double**. You should use a cast if this is not the outcome you want.

Expressions with **long long** or **unsigned long long** operands behave the way a C programmer would expect. The result of such an operation having only integral operands is a **signed** or **unsigned long long** (with a couple of exceptions), **unsigned** if any operand (after promotion) was **unsigned**, and otherwise signed. The main exception to the rule stated above has to do with the shift operator. If the first operand of a shift operand is **long long**, the result will be **long long**, but the type of the second operand has no effect on the type of the result. Thus, 1L << 10LL has type **long**, but 1LL << 10L has type **long long**.

long long bit fields are not supported. **long long** enum values are also not supported.

long long variables are ordinarily aligned on a doubleword boundary. The SAS/C extension **__noalignmem** can be used to have unaligned **long long** members in a structure.

Subscript expressions are always converted to **int**. An expression such as **b[0x100000001LL]** is syntactically valid, but has the same meaning as **b[(int)0x100000001LL]** or, equivalently, **b[1]**, which may not have been the programmer's intent.

The control variable of a switch statement may have **long long** type.

The implementation of **long long** in SAS/C is compatible with the ISO 1999 C Language Standard.

The addition of the **long long** data type is an extension to the ANSI/ISO C language standard of 1989. In most cases, the existence of **long long** does not change the meaning of existing standard-conforming programs. However, in some cases, such changes are possible. These changes can occur mostly because of the potentially different meanings of arithmetic constants. Take the following example:

```
long m = (0xffffffff * 0xffffffff) / 0xffffffff;
```

Prior to SAS/C Release 7.00, the initial value of m was 0. For Release 7.00 and beyond, the value will be -1, because 0xffffffff is now treated as a **signed long long** constant rather than as an **unsigned long** constant.

Another way in which the addition of the **long long** data type could affect existing standard-conforming programs is that certain library headers have been updated to include prototypes for **long long** functions. Any program that uses one of these names for a different purpose and includes the corresponding header file will probably not compile or execute. The compiler **define** option can be used in such a case to define the feature test macro **_SASC_HIDE_LLLIB**, which suppresses these prototypes.

Language Extensions

This section describes extensions to the ISO/ANSI C language implemented by the compiler. Library extensions are described in SAS/C Library Reference, Volume 1 and SAS/C Library Reference, Volume 2. Note that use of these extensions is likely to render a program nonportable.

Embedded $ in identifiers

The dollar sign ($) can be used as an embedded character in identifiers. If the dollar sign is used in identifiers, the **dollars** compiler option must be specified. As mentioned, use of the dollar sign is not portable because the dollar sign is not part of the portable C character set specified by the C Standard. Also, the dollar sign cannot be used as the first character in an identifier; such usage is reserved for the library.

Comment nesting

The compiler optionally allows comments to be nested. (The C Standard does not sanction this usage.) The **comnest** compile-time option must be specified to enact comment nesting. When comment nesting is honored, each **/*** encountered must be matched by a corresponding **-->** before the comment terminates. This feature makes it

easy to comment out large sections of code that contain comments. Thus, sections of debugging code can be removed easily and preserved. Comment nesting is nonportable.

C++ Style Comments

The SAS/C Compiler now supports C++ style line comments. A line comment starts with two forward slashes and goes to the end of the line. An example of the new comment extension is:

```
// This is a comment line
```

Note: This support is turned off if the **strict** compiler option is used. △

Specifying floating-point constants in hexadecimal

An extended format for floating-point constants enables them to be specified in hexadecimal to indicate the exact bit pattern to be placed in memory. A hexadecimal **double** constant consists of the sequence 0.x, followed by 1 to 14 hexadecimal digits. (If there are fewer than 14 digits, the number is extended to 14 digits on the right with 0s.) A hexadecimal **double** constant defines the exact bit pattern to be used for the constant. As an example, 0.x411 has the same value as 1.0. Use of this feature is nonportable.

Function pointer formats

The compiler supports two function pointer formats: local and remote. A remote function pointer is indirect; it points to an 8- or 12-byte area containing the address of the function code as well as the address of the **extern** (and possibly **static**) data associated with the load module. A local function pointer is direct; it is simply the address of the function code. No other addresses are needed. In 370 object code terminology, a local function pointer is a V-type address constant.

The remote format supports pointers to functions in other load modules that have their own set of externs. Since most of the run-time library functions are in separate load modules, library functions that accept function pointer arguments, such as **signal** or **atexit**, typically require remote function pointers.

The local format is simpler. Many assembler language subroutines that accept subroutine addresses only accept addresses in this format. The disadvantage is that a function pointer in local format cannot call a function in another load module if the called function references **extern** or **static** data in that load module, or if that function calls C library routines that might reference such data.

You should use the remote format unless your application has a specific need for function pointers in local format. The remote format is supported by all library functions.

By default, all function pointers are remote. The **__local** and **__remote** keywords explicitly declare function pointers in local or remote format. You can use the **pflocal** option to force the compiler to generate local format function pointers by default.

Far Pointer Support

SAS/C supports the **__near** and **__far** keywords in pointer type declarations as part of its access-register mode support. See “Access Register Mode Support” on page 48 for more information.

Note: The C++ translator does not support **__near** and **__far** for Release 7.00. △

The __local and __remote keywords

The **__local** and **__remote** keywords can be used in function pointer declarations to specify whether the function pointer is in remote or local format.

The **pflocal** compiler option specifies that all function pointers declared in a compilation are local, except those specifically declared with the **__remote** keyword. Under the default, **nopflocal**, most function pointers declared in the compilation are remote, except those specifically declared with the **__local** keyword.

There are three exceptions to this rule: the **__asm**, **__ref**, and **__ibmos** keywords. Use of the **__asm** or **__ref** keywords in a function pointer declaration implies that the declared function pointer is local unless the **__remote** keyword is explicitly specified in the declaration. Function pointers that are declared with the **__ibmos** keyword are always local.

Remote function pointers can be converted to local function pointers. However, local function pointers cannot be converted to remote function pointers. Local function pointers cannot be passed as arguments to library functions.

Below is a list of the library functions that require remote function pointers.

atcoexit

atexit

atfork

bldexit

bsearch

btrace

buildm

cmsrxfn

cosignal

costart

loadd

loadm

qsort

sigdef

signal

unloadd

unloadm

The **sa_handler** field of the library **sigaction** structure is a remote function pointer.

The function pointers that specify alternate **strcoll** and **strxfrm** functions for user-added locales must be remote, and the function pointers that specify a DCB exit routine for the **osdcb** and **osbdcb** functions must be remote.

The address of a function, that is, the value of **&fnc_name**, is either local or remote, depending on the setting of the **pflocal** option. If **pflocal** is used, **&fnc_name** is considered a local function pointer. If **nopflocal** is used, **&fnc_name** is considered a remote function pointer. You can override this behavior by using an explicit cast, as in the following example:

```
atexit((__remote void (*)(void)) &exit_func);
```

Note that you must be careful using function addresses within conditional expressions. In an expression like the following example, both function addresses are converted to the default function pointer type.

```
test ? &fnc_name1 : &fnc_name2
```

If the **pflocal** option was specified and the expression above is assigned to a **__remote** function pointer variable, an error will be indicated by the compiler, since the result of the conditional expression is a local function pointer. You should use casts in expressions of this sort to ensure correct interpretation, as in this example:

```
test ? (__remote void(*)(void)) &fnc_name1 :
       (__remote void(*)(void)) &fnc_name2
```

Note: The SAS/C dynamic-loading functions that are described in SAS/C Library Reference, Volume 2 require the use of remote function pointers. You cannot use the **loadm** function with a function pointer that has been declared with the **__ibmos** keyword. △

Keywords for assembler language functions

SAS/C supports three keywords that can be used to declare functions and pointers to functions written in assembler language that expect a parameter list in OS format:

- **__asm**
- **__ref**
- **__ibmos**

Refer to "__asm, __ref, and __ibmos Keywords" on page 211 for a discussion of these keywords.

__weak storage class modifier

The **__weak** storage class modifier applies only to references to external named objects and functions. For objects, it has meaning only for **__norent** objects or **const** objects that can be initialized at compile time. (See Chapter 3, "Code Generation Conventions," on page 45 for a discussion of **__norent** objects.) The **__weak** keyword is placed next to **extern** on the declaration of those objects and functions. The **__weak** keyword causes the compiler to generate weak references to the declared object. A weak reference suppresses autocall of the symbol by both COOL and the linkage editor or loader. A symbol or function declared with**__weak** need not actually be present in the load module unless specifically included, or referenced by some other compilation in which it is declared without the use of **__weak**. **__weak** does not apply to definitions; therefore, it does not cause the creation of a new storage class. **__weak** external objects are still storage class **extern**.

As a storage class modifier, **__weak** cannot appear as part of a **typedef**, in a cast, on a structure member, and so on. Also, you cannot have a "pointer to **__weak**," any more than one can have a "pointer to **extern**."

When declaring a **__weak** pointer, you must place the **__weak** after the asterisk (*****). The following example demonstrates the use of **__weak**:

```
__weak extern double wd;
extern double * __weak wpd;  /* __weak pointer to double */
__weak int wf();
```

You can use the **isunresolved** library macro to test whether or not a **__weak** reference has been resolved by the linkage editor. See SAS/C Library Reference, Volume 1 for more information about **isunresolved**.

The @ operator

The **@** operator is a language extension provided primarily to aid communication between C and non-C programs.

In C, the normal argument-passing convention is to use call-by-value; that is, the value of an argument is passed. The normal IBM 370 (non-C) argument-passing conventions differ from this in two ways. First, arguments are passed by reference; that is, each item in the parameter list is an argument address, not an argument value. Second, the last argument address in the list is usually flagged by setting the high-order bit, which does not change the value of the address since IBM 370 addresses are 31 bits (XA) or 24 bits (non-XA).

A simplistic approach to the problem of call-by-reference is to precede each function argument by the ampersand (**&**) operator, thereby passing the argument address rather than its value. For example, you can write **asmcode(&x)** rather than **asmcode(x)**. This approach is not generally applicable because it is frequently necessary to pass constants or computed expressions, which are not valid operands of the address-of operator. The compiler provides an option to solve this problem.

When the compiler option **AT** is specified, the at sign (**@**) is treated as a new operator. The **@** operator can be used only in an argument to a function call. (The result of using it in any other context is undefined.) The **@** operator has the same syntax as **&**. In situations where **&** can be used, **@** has the same meaning as **&**.

In addition, **@** can be used on non-lvalues such as constants and expressions. In these cases, the value of **@expr** is the address of a temporary storage area to which the value of **@expr** is copied.

One special case for the **@**operator is when its argument is an array name or a string literal. In this case, **@array** is different from **&array.** While **@array** addresses a pointer addressing the array, **&array** still addresses the array.

The compiler continues to process the @ operator as in earlier releases when the @ is in the context of a function call. Use of **@** is nonportable. Its use should be restricted to programs that call non-C routines using call by reference.

Nesting of #define

If the compiler option **redef** is specified, multiple **#define** statements for the same symbol can appear in a source file. When a new **#define** statement is encountered for a symbol, the old definition is stacked but is restored if an **#undef** statement for the symbol occurs. For example, if the line

```
#define XYZ 12
```

is followed later by

```
#define XYZ 43
```

the new definition takes effect, but the old one is not forgotten. Then, when the compiler encounters the following, the former definition (12) is restored:

```
#undef XYZ
```

To completely undefine XYZ, an additional **#undef** is required. Each **#define** must be matched by a corresponding **#undef** before the symbol is truly forgotten. Identical **#define** statements for a symbol (those permitted when **redef** is not specified) do not stack.

Preprocessor directives for listing control

The following preprocessor commands are available to control the format of the printed listing. They have no effect on any aspect of a program except the program listing and can appear anywhere in program code, except as a continuation line.

- **#pragma eject**
- **#pragma title *text***

- **#pragma space *n***

pragma can be omitted. However, omitting **pragma** renders a program nonportable.

- The **#pragma eject** statement skips to a new page in the listing at the point where **#pragma eject** occurs.
- The **#pragma title *text*** statement stores the title specified by *text* and prints it at the top of subsequent pages of the listing. The text following **title** should be a C string literal. If the text is not a single valid C string literal, it is accepted, but errors may occur if C tokenization rules are not adhered to, for example, if the text contains an unmatched quote. Each time a new **#pragma title *text*** statement is found, any title specified previously is discarded and the new one is used. The **#pragma title *text*** statement does not automatically cause a skip to a new page when it is encountered by the compiler. To skip to a new page and print a new title, the **#pragma eject** statement must follow the **#pragma title *text*** statement.
- The **#pragma space *n*** statement causes a skip of *n* lines (*n* is an integer) in the listing at the point where the statement occurs. If *n* is greater than the number of lines left on a page, the listing skips to the next page and continues on the first line after all the page headings.

Anonymous unions

An anonymous union, that is, a union with no associated identifier, can be declared in a structure. Members of anonymous unions are in the same scope as the containing structure. Here are two examples of anonymous unions:

```
union ANON {
   int i;
   short o[2];
} ;

static struct {
   int a;
   union ANON;
   double d;
}  ex;

static struct {
   int a;

   union {
      int i;
      short o[2];
   } ;
   double d;
}  ex;
```

Member **o[1]**of the union can be accessed by using this expression:

```
ex.o[1]
```

The members of an anonymous union are in the same name space as that of other members in the containing structure. Therefore, a member of the union cannot have the same identifier as a member of the containing structure or a member in another anonymous union in the containing structure.

Other than the above considerations, anonymous unions have the same properties and can be used in the same manner as other unions.

Noninteger bitfields

By default, all bitfields must have type **int**, **signed int**, or **unsigned int**. The **bitfield** compiler option can be used in order for other types to be used. If the **bitfield** option is used, the compiler accepts any integral type in the declaration of a bitfield as in this example:

```
struct {
   char f1 :3;
   signed short f2 :15;
   unsigned long f3 : 28;
} ex;
```

Types that are not **int** can be used to specify the allocation unit to be used by the compiler. (See "Structure and union type names" on page 25.) By default, the allocation unit is an **int**. This means that the compiler allocates 4 bytes of storage for the first bitfield it encounters in a structure definition. Adjacent bitfields are packed into the **int** until not enough bits remain for the next bitfield, a nonbit-field member is declared, or a zero-length bitfield is encountered.

If the **bitfield** option is used, the type of the bitfield determines the allocation unit. If the type is a **char** type, the allocation unit is 1 byte. If the type is a **short** or **long** type, the allocation unit is 2 or 4 bytes, respectively.

The first bitfield declared with a particular type is aligned on the appropriate boundary for that type (as modified by the **bytealign** option, **__noalignmem**, or both). In the previous example, **f1** is allocated in byte 1, **f2** is allocated in bytes 3 and 4, and **f3** is allocated in bytes 5 through 8.

The **bitfield** option also specifies the allocation unit to be used for **int** bitfields. The unit can be **char**, **short**, or **long**. When a bitfield of type **int** is declared, the compiler uses the allocation unit specified by the option.

For example, in the following structure definition, the compiler, by default, allocates 4 bytes of storage for the 8 bits:

```
struct {
   unsigned f1 : 3;
   unsigned f2 : 5;
} ex;
```

However, the **bitfield** option can be used to specify that the allocation unit should be a **char**, which specifies that only 1 byte of storage should be allocated, or a **short**, which specifies that 2 bytes of storage should be allocated.

See Chapter 6, "Compiler Options," on page 101 for information on how to specify the **bitfield** option and the allocation unit.

Zero-length arrays

An array of length 0 can be declared as a member of a structure. No space is allocated for the array, but the following member will be aligned on the boundary required for the array type. Zero-length arrays are useful for aligning members to particular boundaries (to match the format of external data for example) and for allocating varying-length arrays following a structure. In the following structure definition, no space is allocated for member **d**, but the member **b** will be aligned on a doubleword boundary:

```
struct ABC {
   int a;
   double d[0];
   int b;
} ;
```

Zero-length arrays are not permitted in any other context.

The __alignmem and __noalignmem keywords

The **__alignmem** and **__noalignmem** keywords can be used in a structure definition to specify whether members of a structure are to be aligned normally (**__alignmem**) or on byte boundaries (**__noalignmem**). The keywords are associated with the structure tag. Note that the keyword must precede the word **struct** in the structure declaration. For example, in the following structure declaration, member **ex.d** will not be aligned on a doubleword boundary, but it will be allocated at the next available location, a word boundary:

```
__noalignmem struct XYZ {
   int a;
   double d;
} ex;
```

This property can be useful when C structures are used to map existing data areas, such as operating system control blocks, that have fields aligned on boundaries other than those normally associated with the C data types.

The keywords can be used to force alignment even when the **bytealign** compiler option is used or to force byte alignment when the **nobytealign** option is used. See the discussion of **bytealign** in "Option Descriptions" on page 105.

__alignmem and **__noalignmem** propagate to any structure members. For example, in the following structure declaration, the members of **s** will be byte-aligned as well:

```
__noalignmem struct XYZ {
   struct ABC {
      char c;
      float f;
   } s;
   double d;
} ex;
```

The keywords can be used in the declaration of inner structures to change alignment requirements. In the following example, the members of the outer structure are not aligned. The members of the inner structure are aligned.

```
__noalignmem struct XYZ {
   int a;
   short b;
   __alignmem struct ABC {
      int c;
      double d;
   } abc;
   double d;
} ;
```

Special keywords for declarations of non-C functions

The compiler accepts a number of special interlanguage communication keywords in the declarations of functions and function pointers. These keywords indicate that the declared function, or the function pointed to, is written in a language other than the C language. The following keywords are accepted by the compiler:START OF ASIS SECTION

__asm __pascal __pli __cobol __fortran __foreign

END OF ASIS SECTION

Note: If you use any of these keywords other than **__asm**, you must also use the SAS/C Interlanguage Communication Feature. Do not use these keywords if you are using another technique for interlanguage communication. △

Here is an example of a function declaration for a function written in FORTRAN. The function returns a value of type **double**, as follows:

```
double __fortran xyz();
```

The keywords can be used in combination either via a **typedef** or directly. For example, suppose **pasfnc** is a function written in Pascal that returns a pointer to a function written in assembler language. The assembler language function, in turn, returns a value of type **int**. To declare **pasfnc**, you can use a **typedef** as in the following example:

```
typedef __asm int (*asmfp)();
__pascal asmfp pasfnc();
```

Here is an example that does not use a **typedef**:

```
char *(__asm *__pli p())();
```

In this example, a PL/I function returns a pointer to an assembler function (the assembler function returns a pointer to **char**).

Do not use the prototype form of function declaration in declarations containing one of these keywords.

See Chapter 3, "Communication with other Languages," in SAS/C Compiler Interlanguage Communication Feature User's Guide for detailed information on how these keywords can be used.

__inline and __actual storage class modifiers

__inline is a storage class modifier. It can go in the same places as a storage class specifier and can be given in addition to a storage class specifier. If a function is declared as **__inline** and the module contains at least one definition of the function, the compiler sees this as a recommendation that the function be inlined. **__actual** is also a storage class modifier. It can be specified with or without the **__inline** qualifier, but it implies **__inline**. **__actual** is used to specify that the compiler should produce an actual (callable) copy of the function if the function has external linkage. If the function has internal linkage, the compiler may not output an actual function if it does not need one.

For additional information, see the discussion of **__inline** in "Global Optimization Compiler Options" on page 66 and **__actual** in "The __actual Keyword for Inline Functions" on page 74.

The #pragma options statement

The **#pragma options** statement specifies compiler options within program source code. More than one **#pragma options** statement can be used in a source file. See Chapter 6, "Compiler Options," on page 101 for more information about the **#pragma options** statement.

The #pragma linkage statement

The compiler accepts the following statement in which identifier is the name of a function or a typedef of a function. This statement specifies that identifier is called using IBM OS linkage.

```
#pragma linkage (identifier ,OS)
```

Example Code 2.1 on page 37 illustrates the use of the **#pragma linkage** statement in a program.

Example Code 2.1 Sample #pragma linkage Statement

```
extern int asm_func(void);        /* Declare 'asm_func' as called */
#pragma linkage (asm_func,OS)     /*  using OS-linkage.           */
typedef int os_linkage_t(void);   /* Declare functions of type    */
#pragma linkage (os_linkage_t,OS) /* 'os_linkage_t' as called     */
extern os_linkage_t asm_func;     /*  using OS-linkage.           */
```

The compiler accepts the **#pragma linkage** statement to ensure compatibility with IBM products whose C language interface functions are defined with this statement. Refer to the IBM documentation for a specific product for more information. For more information on IBM OS linkage, see Chapter 6, "Compiler Options," on page 101.

The #pragma map statement

The compiler accepts the following statement:

```
#pragma map (external-identifier,"external-name")
```

This statement directs the compiler to use external-name as the object code external symbol for external-identifier. The external identifier must be a C identifier of storage class **extern**. The external symbol can be any sequence of characters, enclosed by double quotes. If the external symbol is longer than eight characters, the compiler truncates it on the right to eight characters.

For example, suppose you had an assembler language module named ABC$DEF and you wanted to reference it in your C programs with a more natural looking name. You could use the following **#pragma** statement to map **abc_def** to ABC$DEF:

```
#pragma map(abc_def, "ABC$DEF")
```

Adhering to the following guidelines ensures that the symbols you create do not conflict with the compiler-generated symbols or symbols defined in the SAS/C Library:

- The first character must be alphabetic or a pound sign (#).
- The last character must be alphanumeric.
- The at sign (@) must not be used.
- If the **norent** option has been specified and the external symbol is eight characters or more in length, the first character must be an uppercase alphabetic character.

The compiler issues a warning diagnostic message for any external symbol name that does not follow these guidelines. Depending on the context, the compiler may refer to the external identifier by the external symbol name in a diagnostic message. Appendix 7, "Extended Names," on page 405 contains information on using **#pragma map** with the **extname** compiler option.

Character and String Qualifiers

Release 6.50 introduced **A** and **E** qualifiers for character and string constants. The new qualifiers causes the string to be either ASCII or EBCDIC.

A string literal prefixed with **A** is parsed and stored by the compiler as an ASCII string. An example of its usage is:

```
A"this is an ASCII string"
```

A string literal prefixed with **E** is parsed and stored by the compiler as an EBCDIC string. An example of its usage is:

```
E"this is an EBCDIC string"
```

The translation between ASCII and EBCDIC is based on IBM Code Page 1047 for EBCDIC and ISO 8859–1 (Latin 1) for ASCII.

Implementation-defined Behavior

In many instances, the ISO/ANSI Standard gives an implementor of the C language the freedom to choose an appropriate approach for a particular aspect of the language. The only requirement is that the choice made is explained to the user. For SAS/C software, implementation-defined behavior refers to aspects of the compiler where coding decisions have been made as a result of the SAS/C implementation of the ISO/ANSI Standard C language. The following sections follow the ISO/ANSI Standard in covering implementation-defined behavior for the compiler. The ISO/ANSI Standard conventions are given in Annex G.3 of the ISO/IEC 9899:1990 Standard for Programming Languages – C. The relevant sections of this appendix are noted next to the topic.

Translation (G.3.1)

- Compiler diagnostics are listed in *SAS/C Software Diagnostic Messages*.

Environment (G.3.2)

- The compiler conforms to the ISO/ANSI Standard for a hosted environment as documented in Section 2.1.2.2 of the *American National Standard for Information Systems - Programming Language C*. It also conforms to the environment variables extension documented in Chapter 9, “Run-Time Argument Processing,” on page 185.
- The library recognizes the terminal as the only interactive device. Under OS/390 batch, the terminal is logically represented by the DDname SYSTERM.

Identifiers (G.3.3)

- The number of significant initial characters in an identifier without external linkage is 31. The significant initial characters are **`65,535`** if the **`extname`** option is specified.
- The compiler recognizes eight significant characters in an identifier with external linkage if the **`extname`** option is not specified.
- Case distinctions are not significant in an identifier with external linkage unless the **`extname`** option is specified.

Characters (G.3.4)

- The source and execution character sets include all EBCDIC characters.
- Two shift states are used for encoding multibyte characters; a single byte shift state and a double byte shift state. 0x0e causes entry into the double byte shift state, and 0x0f causes return to the single byte shift state.
- There are eight bits in a character in the execution character set.
- The mapping of members of the source character set (in character constants and string literals) to members of the execution character set is standard EBCDIC. Refer to Table 2.3 on page 23.
- All characters in the basic (or extended) source character set are also in the basic (or extended) execution character set.
- An integer character constant that contains more than one character is interpreted as a 4-byte integer. If it contains more than four characters, only the rightmost four characters are used. A wide character constant that contains more than one character is interpreted by ignoring all characters but the rightmost.
- By default, a non-DBCS locale is used to convert multibyte characters to wide characters (and in all other processing of multibyte characters within the compiler). This can be changed by supplying an environment variable to the compiler that affects its locale. (See Chapter 10, "Localization Functions," and Chapter 11, "Multibyte Character Functions," in *SAS/C Library Reference, Volume 2* for more information.) If the specified locale supports DBCS, EBCDIC DBCS rules are used to convert multibyte characters to wide character codes.
- A plain **`char`** is identical to an unsigned character.

Integers (G.3.5)

- Refer to Table 3.1 on page 46 for the representations and sets of values of the various types of integers. The IBM architecture and SAS/C software use two's complement representation.
- When an integer is converted to a shorter signed integer, the high-order bytes are discarded. The bit pattern of the remaining bytes is unchanged. The bit pattern of an unsigned integer is not changed when it is converted to a signed integer of equal length.
- The following list covers the results of bitwise operations on signed integers:

  ~ (bitwise NOT) — The bits are inverted, that is, 1 bits are set to 0, and 0 bits are set to 1.

  >> (shift right) — The bits are right shifted. The sign bit (bit 0) is used to fill the vacated bit positions on the left.

  << (shift left) — The bits are left shifted. The vacated bit positions on the right are filled with 0s.

  & (bitwise AND) — If the corresponding bits in both operands are 1, then the corresponding bit in the result is set to 1; otherwise it is set to 0.

  | (bitwise OR) — If either of the corresponding bits in the operands are 1, then the corresponding bit in the result is set to 1; otherwise it is set to 0.

  ^ (bitwise XOR) — If the corresponding bits in the operands are not alike, then the corresponding bit in the result is set to 1; otherwise it is set to 0.

- In integer division, the remainder has the same sign as the dividend, except that a 0 remainder is always positive. (These are the results produced by the 370 DIVIDE instruction.)
- In a right shift of a negative-valued signed integral type, the sign bit is used to fill the vacated bit positions on the left; that is, the result retains the sign of the first operand.

Floating point (G.3.6)

- Table 3.1 on page 46 provides the representations and sets of values of the various types of floating-point numbers.
- The IBM 370 representation of a **`double`** can represent all integral values exactly, so no rounding is necessary when an integer is converted to a **`double`**. When a **`long`** integer value is converted to **`float`**, the value is rounded.
- Rounding (away from 0 when either direction is equally correct) occurs when a floating-point number is converted to a narrower floating-point number. (These are the results of the IBM 370 LOAD ROUNDED instruction.)

Arrays and pointers (G.3.7)

- The type of integer required to hold the maximum size of an array (for example, the type of the result of the **`sizeof`** operator, **`size_t`**) is **`unsigned int`**.
- When a pointer is converted to integer, the integer contains the logical address of a data object. (Function pointers may use an indirect representation. See "Local Function Pointers" on page 51 for more information.) If the logical address is not representable in the destination integral type, the conversion is analogous to the conversion from **`unsigned long`** to the destination type. Nonaddress bits in the pointer will be preserved.
- When an integer is converted to a pointer, the value used is that of the integer converted to **`unsigned long`**. The pointer contains the logical address corresponding to the integral value. Nonaddress bits are preserved.
- **`ptrdiff_t`** is **`signed long`**.

Registers (G.3.8)

- There can be up to six integer or pointer register variables and up to two floating-point register variables. The **`register`** keyword is ignored when the **`optimize`** option is used because registers are allocated to variables by the compiler when the **`optimize`** option is specified. The number of registers allocated can be controlled by the **`greg`** and **`freg`** options. See "Optimizations" on page 64 for more information.

Structures, unions, enumerations, and bitfields (G.3.9)

- If a member of a **`union`** object is accessed using a member of a different type, the result is undefined.

- Each element is aligned according to Table 3.1 on page 46 unless the **bytealign** option is used or the **__noalignmem** keyword is used. Padding is introduced as necessary to maintain correct alignment. For bitfields, see "Noninteger bitfields" on page 34. By default, bitfields are aligned on word boundaries.
- Plain **int** bitfields are treated as **unsigned int** bitfields.
- The order of allocation of bitfields within an **int** is left to right.
- A bitfield can straddle a storage unit boundary if the **bitfield** option specifies a storage allocation unit size of less than 4 bytes.
- The values of an enumeration type are represented as **ints**.

Qualifiers (G.3.10)

- Any reference to an object that has **volatile**-qualified type is considered an access to that object.

Declarators (G.3.11)

- There is no limit on the number of declarators that can modify an arithmetic, structure, or union type.

Statements (G.3.12)

- There is no limit to the number of **case** values in a **switch** statement.

Preprocessing directives (G.3.13)

- A value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. However, such a character cannot have a negative value.
- The method for locating includable source files is outlined in Chapter 3, "Code Generation Conventions," on page 45.
- The support of quoted names for includable source files is outlined in Chapter 3, "Code Generation Conventions," on page 45.
- The mapping of source file character sequences is found in Chapter 3, "Code Generation Conventions," on page 45.
- The behavior of each recognized **#pragma** directive is covered earlier in this chapter.
- The compiler does not create the **__DATE__** and **__TIME__** macros when the date and time are not available.

Library functions (G.3.14)

- The null pointer constant to which the macro **NULL** expands is 0.

- For information about the behavior of the **assert** function see Chapter 6, "Function Descriptions" in *SAS/C Library Reference, Volume 1*. The diagnostic printed by the **assert** diagnostic control function is

  ```
  Assertion failed: expr
  Interrupted while: Executing line <number > of <source file >
  ```

  where *number* is the current value of **__LINE__** and *source file* is the current value of **__FILE__**.
- The sets of characters tested for by the **isalnum**, **isalpha**, **iscntrl**, **islower**, **isprint**, and **isupper** functions are described in Chapter 6, "Function Descriptions," in *SAS/C Library Reference, Volume 1* .
- The values returned by the mathematical functions on domain errors are covered in Chapter 6, "Function Descriptions," in *SAS/C Library Reference, Volume 1*.
- On underflow range errors the mathematical functions set the integer expression **errno** to the macro **ERANGE**.
- A value of 0 is returned when the **fmod** function has a second argument of 0.
- The following items are documented in Chapter 5, "Signal-Handling Functions" in *SAS/C Library Reference, Volume 1*:
  - the set of signals for the **signal** function
  - the semantics for each signal recognized
  - default handling and handling at program start-up for each signal recognized
  - the treatment of a signal on entry to a handler
  - whether default handling is reset if the SIGILL signal is received by a handler specified to the **signal** function.
- The last line of a text stream requires a terminating new-line character, unless the stream is an USS HFS file. One is generated if not explicitly written.
- Depending on file format, space characters that are written out to a text stream immediately before a new-line character may or may not appear when read in.

  Note: See Chapter 3, "I/O Functions" in *SAS/C Library Reference, Volume 1* for additional information concerning the **pad** amparm. △
- As many null characters can be appended to data written to a binary stream as it takes to fill out a complete logical record.
- When a file is opened with open mode "a" or "ab", it is initially positioned to the end of file. When opened with "a+" or "a+b", it is initially positioned to the start of the file.
- Whether a write on a text stream causes the associated file to be truncated beyond that point depends on the file type and options. See Chapter 3, "I/O Functions" in *SAS/C Library Reference, Volume 1*.
- For the characteristics of file buffering, refer to the discussion of **setbuf** and **setvbuf** in Chapter 6, "Function Descriptions" in *SAS/C Library Reference, Volume 1*.
- Refer to the section "IBM 370 I/O Concepts" in Chapter 3, "I/O Functions" in *SAS/C Library Reference, Volume 1* for information on the existence of a zero-length file.
- Refer to "Opening Files" in Chapter 3, "I/O Functions" in *SAS/C Library Reference, Volume 1* for the rules for composing valid filenames.
- The terminal can be opened multiple times, as can files open for input only. Opening a nonterminal file for output multiple times or for both input and output gives unpredictable results.
- The effect of the **remove** function on an open file is unpredictable.

- **rename** will fail if a file with the new name exists prior to a call to the **rename** function.
- The output for %p conversion in the **fprintf** function is the pointer value in hexadecimal.
- The input for the %p conversion in the **fscanf** function is a hexadecimal string, optionally preceded by 0x or 0p.
- In the **fscanf** function, the hyphen (-) character has no special meaning. It is interpreted the same way as any other character in the scan list.
- Refer to the documentation for **fgetpos** and **ftell** in Chapter 6, "Function Descriptions" in *SAS/C Library Reference, Volume 1* for information on failure of these functions.
- The messages generated by the **perror** function are listed in *SAS/C Software Diagnostic Messages*.
- **calloc**, **malloc**, and **realloc** return a **NULL** pointer if the size requested is 0.
- Open files are not closed if the **abort** function is called. The effect on temporary files is unpredictable and system dependent.
- If the value of the argument to **exit** is other than 0, **EXIT_SUCCESS**, or **EXIT_FAILURE**, the status returned by **exit** is passed unchanged to CMS or reduced modulo 4096 in OS/390. Under the USS shell, a negative exit status or one greater than 255 is set by the USS kernel to 255.
- See the documentation for the **getenv** function in Chapter 6, "Function Descriptions," in *SAS/C Library Reference, Volume 1* for information on the set of environment names and the method for altering the environment list.
- See the documentation for the **system** function in Chapter 6, "Function Descriptions," in *SAS/C Library Reference, Volume 1* for information on the contents and mode of execution of the string involved.
- The contents of the error message strings returned by the **strerror** function are the same as the **perror** messages.
- The local time zone can be defined by setting the TZ environment variable. See the description of **tzset** in Chapter 6, "Function Descriptions," in *SAS/C Library Reference, Volume 1*. If TZ is not set, the offset from Greenwich time is determined by the operating system, and no Daylight Savings Time information is available.
- The era for the **clock** function under OS/390 is set at the first call to **clock**, under CMS, at virtual machine log on.

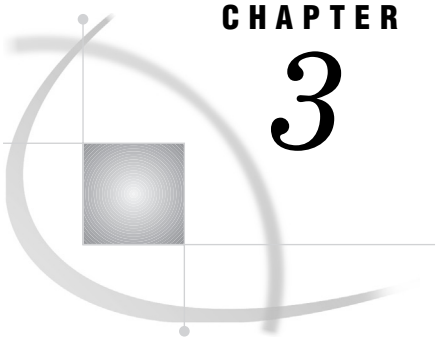
Locale-specific behavior (G.4)

- The execution character set contains all EBCDIC characters in all locales.
- The direction of printing is left-to-right.
- A period (.) is the decimal point character.
- See Chapter 3 in *SAS/C Library Reference, Volume 1* for the documentation of the **ctype** functions for the implementation-defined aspects of character-testing and case-mapping functions.
- The collation sequence for EBCDIC characters is used as the collation sequence of the execution character set.
- SAS/C software uses the normal U.S. time and date conventions for the “C” locale. The exact lisitngs are shown in Table 2.6 on page 44 (both locales are the same).

Table 2.6 C Locale U.S. Date/Time Conversions

Type	Values	
Abbreviated weekday name	Sun	Thu
	Mon	Fri
	Tue	Sat
	Wed	
Full weekday name	Sunday	Thursday
	Monday	Friday
	Tuesday	Saturday
	Wednesday	
Abbreviated month name	Jan	Jul
	Feb	Aug
	Mar	Sep
	Apr	Oct
	May	Nov
	Jun	Dec
Full month name	January	July
	February	August
	March	September
	April	October
	May	November
	June	December
Date representation	Mnn dd yyyy	
	May 01 1989	
Time representation	hh:mm:ss	
	01:22:45	

Note that conventions for user-written locales are defined by the user.



CHAPTER

3

Code Generation Conventions

Introduction

Many important features of the C language are implementation-dependent since the ISO/ANSI Standard language definition does not completely specify all aspects of the language. This flexibility in the finer details permits C to be implemented on a variety of machine architectures without forcing code generation sequences that are elegant on one machine and awkward on another. This chapter focuses on hardware and operating-system-dependent features of the compiler as implemented on machines with IBM 370 architecture that are running under one of the OS/390 or CMS operating systems or any extended architecture system.

Instruction Set

The compiler generates code that uses the standard IBM 370 or 390 instruction set. Therefore, the code will execute on any IBM 370 or 390 architecture machine, including the 309X, 308X, 9370 series, 4300 series, 303X, and the 370/168. Certain floating-point operations and the run-time library routines for conversion of floating-point data require the Extended Precision Floating-Point Feature (or a software simulation).

Arithmetic Data Types

The compiler implements the standard C data types as follows:

- **`doubles`** and **`long doubles`** are 8 bytes in length and are aligned on an 8-byte (doubleword) boundary. **`floats`** are 4 bytes in length and are aligned on a 4-byte boundary.
- **`longs`** and **`ints`**, both signed and unsigned, are 4 bytes in length and are aligned on a 4-byte (fullword) boundary.
- **`shorts`** and **`unsigned shorts`** are 2 bytes in length and are aligned on a 2-byte (halfword) boundary.
- both **`signed`** and **`unsigned char`** are 1 byte (8 bits) in length and are aligned on byte boundaries.
- A structure or union is aligned according to the strictest alignment requirement of any of its members (including other structures or unions). This rule is applied recursively. Note that if you specify the **`bytealign`** compiler option, most data items, including all those in structures, are generated with only character alignment.

All data types use the normal 370 representation. Table 3.1 on page 46 summarizes the characteristics of the arithmetic data types.

Table 3.1 Data Type Characteristics

Type	Length	Alignment	Range
`char`	1	byte	0 to 255 (EBCDIC character set)
`signed char`	1	byte	— 128 to 127
`unsigned char`	1	byte	0 to 255 (EBCDIC character set)

Type	Length	Alignment	Range
short	2	halfword	— 32768 to 32767
unsigned short	2	byte	0 to 65535
int	4	word	— 2147483648 to 2147483647
long	4	word	— 2147483648 to 2147483647
unsigned	4	word	0 to 4294967295
float	4	word	=/ — 5.4E — 79 to =/ — 7.2E75
double	8	doubleword	=/ — 5.4E — 79 to =/ — 7.2E75
long double	8	doubleword	=/ — 5.4E — 79 to =/ — 7.2E75
char unsigned char	defines an 8–bit unsigned integer		
signed char short short int	defines an 8–bit signed integer defines a 16-bit signed integer		
unsigned short unsigned short int	defines a 16–bit unsigned integer		
int long long int	defines a 32–bit signed integer		
unsigned unsigned int unsigned long int	defines a 32–bit unsigned integer		
float	defines a 32–bit signed floating-point number in the standard 370 representation, that is, a sign bit, a 7-bit biased hexadecimal exponent, and a 24-bit fractional part. The exponent bias is 64. All constants and results generated by compiled code are normalized (except for constants specified in hexadecimal notation). This representation is equivalent to approximately 6 or 7 decimal digits of precision		
double long double	defines a 64–bit signed floating-point number in the standard 370 representation, that is, a sign bit, a 7-bit biased hexadecimal exponent, and a 56-bit fractional part. The exponent bias is 64. All constants and results generated by compiled code are normalized (except for constants specified in hexadecimal notation). This representation is equivalent to approximately 16 or 17 decimal digits of precision		

Note that in contrast to the signed integer representations, negative floating-point values are not represented in two's complement notation; positive and negative numbers differ only in the sign bit. Also note that there are multiple representations of 0; any value that has a 0 fractional part is treated as 0 by the IBM 370 floating-point instructions, regardless of the exponent value.

Code that checks **float** or **double** objects for 0 by means of type punning (that is, examining the objects as if they were **int** or some other nonfloating-point type) may assume (falsely) negative 0 not to be 0.

Arithmetic Exceptions

All arithmetic operations are performed by inline code. The code expects fixed-point overflow to be disabled (with the PSW mask bit), causing integer overflow to be ignored for both unsigned and signed operands. Integer division by 0 causes abnormal program termination (program interruption code 0009) if no arithmetic signal handler is defined.

Floating-point exceptions produce a program interrupt that causes abnormal program termination (program interruption codes 000D or 000F) if no arithmetic signal handler is defined. Note that the code will execute correctly whether or not floating-point underflow is disabled (via the PSW mask bit). Consult the appropriate IBM principles of operation manual for more information about the floating-point formats and exceptions.

Data Pointers

Pointers to all data types are 4 bytes long and are aligned on a 4-byte (fullword) boundary unless the **bytealign** compiler option or **__noalignmem** keyword is used. All pointers to data have the same representation. (However, code that relies on this fact is likely to be nonportable.) **NULL** has all 32 bits set to 0.

The rest of this section is relevant only if you need to know the details of pointer representation or if you plan to create pointers in non-C code and pass them to C code.

IBM 370 and XA addressing do not use all 32 bits of a 4-byte pointer. In 24-bit addressing mode, only bits 8 through 31 (starting at the left) are used as an address. In 31-bit addressing mode (provided only on XA machines), only bits 1 through 31 are used as an address (bit 0 is not). All addresses generated by the compiler have the nonaddress bits (bits 0 through 7 or bit 0) set to 0. Therefore, all pointers that are set to an address by C code (whether at compile time or execution time) have the nonaddress bits set to 0.

In most circumstances, it does not matter whether the nonaddress bits of a pointer are 0. The bits are ignored by the hardware when the pointer is used to address data. However, if pointers are compared, the setting of the bits becomes significant. If you create pointers in non-C code and pass them to C code, be sure that the nonaddress bits are 0, or turn them off (using a cast) in C code before using the pointer in comparison operations.

The value in the nonaddress bits also becomes significant if pointers are subtracted from each other. In this case, the code generated by the compiler always clears the nonaddress bits to 0 before doing the subtraction, so this case need not concern you.

Finally, note that pointer assignment, pointer addition, and pointer-integer subtraction do not set the nonaddress bits to 0. If these operations take a pointer created outside of C as input and that pointer has nonaddress bits set, then the result may also have nonaddress bits set (not necessarily the same bits).

To summarize, the compiler always creates pointers with the nonaddress bits all set to 0 but may propagate non-zero values in these bits from pointers created outside of C.

Access Register Mode Support

As part of its systems-programming support, SAS/C allows a C function to execute in access register mode, and to access data in dataspaces or other address spaces. This facility is presently supported only in an OS/390 SPE environment (see Chapter 14, “Systems Programming with the SAS/Compiler” in the SAS/C Compiler and Library User’s Guide). See the IBM ESA/390 Principles of Operation publication (SA22-7201) for more information on access registers and access register mode.

Access to data in a dataspace or another address space is accomplished through the use of far pointers. A far pointer is an 8-byte object that contains an ALET (an address space identifier) as well as an address. When a far pointer is dereferenced, the ALET is loaded into the hardware access register corresponding to the general register containing the address part of the pointer. The ALET is stored in the first word of the far pointer, and the address in the second word.

Note: When a regular (near) pointer is dereferenced, an ALET of 0, specifying the primary address space, is loaded into the appropriate access register. △

Far pointers are declared using the **__far** keyword. This keyword is syntactically similar to the **const** keyword, but applies only to data pointer types. The **__near** keyword can be used to explicitly declare a pointer as a near pointer. Here are some examples of declarations using the **__near** and **__far** keywords.

__far char *keyptr;
keyptr is a far pointer to character.

__far char **nearfar;
nearfar is a near pointer to a far pointer to character.

char *__far *farnear;
farnear is a far pointer to a near pointer to character.

int cache(__near void *, __far void *);
cache is a function taking two arguments — a near pointer and a far pointer.

__far struct udata *lookup(char *uname);
lookup is a function that returns a far pointer to a structure. Its argument is a near pointer to character.

The following uses of **__far** will fail to compile for the indicated reason:

```
__far double[40000];  /* Error: __far is only meaningful for pointers */
int (__far *service)(int);
                     /* Error: only data pointers, not function
                        pointers, can be far */
```

The result of the address operator (&) is normally a near pointer. However, the address of an object accessed via a far pointer is itself a far pointer. For example, **&(a->b)** is a near pointer if **a** is near, and a far pointer if **a** is far. This distinction is important mostly when passing arguments to functions without prototypes, or with a variable number of arguments.

Far pointers may be converted to near pointers and vice versa. When a far pointer is converted to near, its ALET is discarded; when a near pointer is converted to far, a zero ALET is implied. Note that converting from a far pointer to a near pointer is ordinarily "safe" only when its ALET is known to be 0.

Far pointers may be subtracted. The result is a **ptrdiff_t**, and the ALET portions of the pointers are ignored during the subtraction. Because far pointers are 8-byte pointers, they can be converted to a **long long** integer without loss of information. Conversion to a shorter integer type will cause the ALET information to be lost.

The **armode** compiler option is required for any C function that uses the value of a far pointer. Code that is compiled with the **armode** option runs entirely in access register mode, except that function entry, function exit, and subroutine calls take place in primary address space mode. This allows code compiled with the **armode** option and code compiled without it to be freely mixed.

The SAS/C compiler's access register mode support uses a callee saves convention for the access registers. This means that a called access register mode routine saves the access registers on entry and restores them on exit. Note that this implies that any assembler language routine that might be called, directly or indirectly, from a caller compiled with the **armode** option must restore any access registers it uses before returning.

When a far pointer is passed to a function, the pointer is doubleword-aligned in the argument list. (Far pointers in static or auto storage, on the other hand, need only be fullword-aligned.) When a function returns a far pointer, the address is returned in general register 15 and the ALET in access register 15.

Note that obtaining access to data in dataspaces or other address spaces remains the responsibility of the user program. Stack and heap memory, as well as control blocks like the PRV are always allocated in the primary address space. The library routines **dspserv**, **aleserv**, and **falloc**, which are described in SAS/C Library Reference, Volume 2, can be used to create a dataspace and to allocate space in a dataspace. This functionality can also be performed in assembler language.

Function Pointers

SAS/C software allows two types of function pointers, remote and local. Function pointers of each type may be declared using the nonstandard keywords **__remote** and **__local**. Function pointers are remote by default. Normally, you do not need to be concerned with the details of function-pointer implementation. If all of your programs are in C, the compiler takes care of setting up the pointer and calling a function through the pointer. This section provides information for users who have to build or use C function pointers in assembler language routines, who have to map between function pointers and data pointers, or who need to know the details of function-pointer implementation for some other purpose.

Remote Function Pointers

By default, all function pointers are remote. You can also explicitly declare a remote function pointer using the positional type qualifier keyword **__remote**. For example:

```
int __remote (*remote_fp)(void);
```

Remote function pointers are 4 bytes long and are aligned on a fullword boundary. They are indirect, that is, they do not point directly to executable code. Remote function pointers can call a function in another load module which has a different pseudoregister vector. A remote function pointer must define both the address of the called function and the new pseudoregister vector. An indirect implementation makes it possible for remote function pointers to contain both pieces of data. A remote function pointer points to an object containing two 4-byte addresses. The first address is the address of the function, the second is the address of the pseudoregister vector for the load module containing the function. For more information, refer to “External Variables” on page 55.

Use the following procedure to call a function through a remote function pointer in assembler language:

1. Load the function pointer into general register 15.
2. Save the value in CRABPRV in a general register.
3. Copy the second fullword addressed by general register 15 (the new pseudoregister vector address) into CRABPRV. If the called function is in the same load module as the calling function, the new address will be the same as the old address, but it is quicker to move it than it is to check.
4. Load the address in the first fullword (the called function address) into general register 15 and call the function with a standard BALR 14,15.
5. When the called function returns, restore the old value of CRABPRV from the general register in which it was saved.

CRABPRV is a field containing the address of the pseudoregister vector for the currently executing load module. During program execution, this field is always located at offset decimal 12 from the address in general register 12.

The following example illustrates this procedure:

Example Code 3.1 Sample Assembler Language Routine for Calling a Function with a Remote Function Pointer

```
L    R15,FUNCPTR       Load function pointer into R15.
L    RX,12(,R12)       Save current pseudoregister vector
                       address. This assumes that R12 has
                       the CRAB address.
MVC  12(4,R12),4(R15)  Copy new pseudoregister vector
                       address into CRABPRV.
L    R15,0(,R15)       Load function address into R15.
BALR R14,R15           Call the function.
ST   RX,12(,R12)       Restore old pseudoregister vector
                       address.
.
.
.
FUNCPTR  DS   A
```

If you are using the **norent** compiler option, the format of a remote function pointer that has not yet been used to call a function is slightly different. A function pointer in this state addresses an object containing three addresses. The first is the address of a library routine (L$CFNAD), the second addresses the object itself, and the third addresses the entry point of the function. The procedure for calling a function through a function pointer in this format is no different from that outlined above. Steps 2, 3, and 5 (the psuedoregister vector address swap) must be performed. You can identify a function pointer in this format by testing bit 0 of the second address. If the bit is set to 1, the function pointer is in this format.

Local Function Pointers

Local function pointers are 4 bytes long and aligned on a fullword boundary. A local function pointer points directly to the entry point of the function.

To declare a local function pointer, specify the keyword **__local** preceding the open parenthesis of the function pointer declaration, as in the following example:

```
int __local (*local_fp)(void);
```

Earlier versions of SAS/C documentation showed the keyword **__local** preceding the return type. That syntax is ambiguous, but the compiler attempts to honor it in simple cases. The following example is a correct declaration of a local function pointer that returns a remote function pointer.

```
int __remote (*__local(*local_fp)(void))(int);
```

The above declaration could not be specified using the previously documented syntax without use of a **typedef**. Nevertheless, in practice, a **typedef** would be recommended for clarity.

The **pflocal** compiler option can be used to specify that all function pointers in a program are local except for those specifically declared with the **__remote** keyword. For more information on the **pflocal** option, see Chapter 6, “Compiler Options,” on page 101.

In assembler language terminology, the pointer can be thought of as a V-type address constant. To call a function through a local function pointer in assembler language, load the value of the function pointer into the appropriate general register, usually general register 15, and call the function with the instruction sequence expected by the function. Example Code 3.2 on page 52 illustrates the standard calling sequence.

Example Code 3.2 Sample Assembler Routine for Calling a Function with a __local Function Pointer

```
L    R15,FUNCPTR     Load __local function pointer into R15.
BALR R14,R15         Call the function.
ST   R15,RETVAL      Save function return value.
.
.
.
FUNCPTR  DS   A
```

Function pointers declared using the **__asm**, **__ref**, or **__ibmos** keyword are local by default. You can, however, use the **__local** keyword in such a declaration without causing an error.

Conversions

It is possible to convert a remote function pointer to a local function pointer by assignment or with a cast. This can be used to obtain the entry point address of a C function or to pass the entry point address to an assembler language routine. For instance, the code in Example Code 3.3 on page 52 passes the actual entry point address of **vdefexit** to **ISPLINK**.

Example Code 3.3 Conversion of a Remote Function Pointer to a Local Function Pointer

```
#include <stddef.h>
extern int cuserxt();
extern __asm int ISPLINK();         /* Declare ISPLINK as an        */
                                    /* assembler language function. */

void main()
{
   struct {
      __local int (*exit)();
      char *data;
   } udata;
   char name[41] ;
   .
   .
   .
      /* Assign the cuserexit function pointer */
      /* to the __local function        */
      /* pointer udata.exit.                  */

   udata.exit = cuserxt;
   udata.data = NULL;

      /* Define an ISPF variable as using a user exit. */
   (void) ISPLINK("VDEFINE ", "NAME ", name, "USER ",
                  @40, " ", @udata);
   .
   .
   .
}
```

Since local function pointers do not contain a pseudoregister address, conversion from local to remote is not possible. You can frequently use the **buildm** function to create a remote function pointer from a local function pointer. For more information on the **buildm** function, see Chapter 1, "Dynamic-Loading Functions," in SAS/C Library Reference, Volume 2 .

Note: Never attempt to use a function pointer to modify the code of a function. In addition to the reentrancy implications, note that a **__remote** function pointer does not address the function's code directly, and using it to store new data will produce unpredictable results. △

For more information about using assembler language routines in C programs, see Chapter 11, "Communication with Assembler Programs," on page 209. For more information about the **__remote** and **__local** keywords, refer to Chapter 2, "Source Code Conventions," on page 9.

Compiler-generated Names

During compilation, the compiler creates names for various data objects in the compilation. In general, compiler-generated names are based on the section name. The section name, in turn, can be specified by the **sname** option or determined by default.

In general, compiler-generated external names are created by appending one or more special characters to the section name. Each type of data object has a unique special character associated with it. If the section name is less than seven characters long, then all of the created names are suffixed by an **@**, followed by the special character for the data object type (unless that is a second **@**). If the section name is exactly seven characters long, then only the special character is used as the suffix.

Control Section Names

The compiler creates one or more control sections (CSECTs) for a compilation. Each CSECT contains a specific type of data; for example, there is a separate CSECT for the executable code for the functions in the compilation. The number of CSECTs created varies depending on the compilation and the compiler options used. Table 3.2 on page 53 lists the possible CSECT suffixes and compiler options that may cause the CSECT to be created.

Table 3.2 Control Section Suffixes

Suffix	Type of Data	Compiler Options
@	executable code	(any)
:	constants	(any)
$	string literals	(any)
$	static data	**norent, rentext**
=	initialization data	**rent, rentext**
?	line number/offset table	**lineno, debug**
+	run-time constants	(always generated)

Suffix	Type of Data	Compiler Options
>	extended funtion names	**extname**
<	extended identifiers other than function names	**extname**

Run-time constants

The run-time constants CSECT contains data items used by the library or the debugger during program execution. The program itself, or another program, such as a dump analysis program, may also refer to the data in the run-time constants CSECT.

The following structure definition shows how the compiler stores the run-time constants in this CSECT:

```
struct Run_Time_Constants {
   int RESERVED1;
   int RESERVED2;
   void *RESERVED3;
   void *ext_names;
   char datetime1[16] ;
   char sname[8] ;
   time_t datetime2
   void *statics;
   int RESERVED4;
   int dbg_filename_len;
   char dbg_fileneame[1];
};
```

The **ext_names** field is a pointer to the extended function names CSECT or is **NULL** if an extended function names CSECT does not exist. The **datetime1** field is the compilation date and time in character format. The date and time field is 16 bytes long and is in the following format:

```
ddMMMyy hh:mm:ss
```

An example of the format of the date and time field follows:

```
19DEC90 12:34:56
```

Under OS/390, the **sname** field contains the section name of the compilation and is terminated by a 0 byte. Under CMS, the **sname** field contains the filename of the source file. The filename is not terminated with a 0 byte if it is exactly eight characters long. The **datetime2** field is the compilation date and time in numeric **time_t** format. For more information on the type **time_t** see "Timing Functions," in Chapter 2 of SAS/C Library Reference, Volume 2. The **statics** field is the pointer to the static data or string literal CSECT, that is, the CSECT with a name ending in a dollar sign ($).

Note: All of the fields in this structure definition are reserved for use by the compiler, library, or debugger and should not be modified. Modification may result in unpredictable results. △

At runtime, the address of the run-time constants CSECT is at offset +8 in the constants CSECT. General register 4 always contains the address of the constants CSECT at execution time. The address of the constants CSECT is also stored at decimal offset 36 from the start of each function.

OMD370 disassembles the run-time constants CSECT when the **verbose** option is specified.

Extended names CSECTs

The CSECTs for extended function names and extended identifiers other than function names are created when the **extname** option is specified. See "External Variables" on page 55 for a detailed discussion of the extended names CSECTs. OMD370 displays the extended names CSECTs when the **verbose** option is specified.

Exceptions

When the **norent** option is used, function pointers are defined as CSECTs. If the function name is seven or fewer characters long, the CSECT name is created by prefixing an ampersand (**&**) to the function name. If the function name is longer than seven characters, the compiler generates a special name, only distantly related to the actual name of the function, beginning with an unusual character.

In addition to the names described above, the compiler may generate other CSECTs or pseudoregisters that do not follow the same naming convention. Typical examples are @EXTERN#, the CSECT containing initialization data for external variables stored in pseudoregisters; and @ISOL@, used for a CSECT when no section name can be determined. (This occurs only when the compilation contains no externally visible functions or data.)

Pseudoregister Names

When the compiler options **rent** and **rentext** are used, the compiler creates pseudoregisters to contain certain types of external and static data (see "External Variables" on page 55). The names of the pseudoregisters are created as described in the previous sections. The table below lists the possible pseudoregister suffixes

Table 3.3

Suffix	Type of Data
*	static data other than function pointers
&	static function pointers

The const Type Qualifier

If either the **rent** or **rentext** compiler option is used, defined external and static objects that are qualified as **const** are placed in the string literal CSECT if possible. See "Placement of Data" on page 58

External Variables

External identifiers differ from ordinary identifiers in one important respect: IBM 370 link utilities treat upper- and lowercase letters as if they were the same. Therefore, when default options are used, the compiler converts all external identifiers to uppercase. For example, although the programmer may consider **vermont** and **VERMONT** to be two different functions, the linkage editor does not. If they are intended to be distinct functions, the compiler **extname** option should be used. (See "The extname option" on page 56.)

External names are limited by the 370 object code format to eight characters, and SAS/C translates underscores (_) in external names to pound signs (**#**). Since the compiler always assumes that external names have the same characteristics as ordinary identifiers, programmers must be careful not to define external names that the compiler believes are different but that the linkage editor interprets as the same name.

A safe rule is to use lowercase letters only for all items that are declared external or that have no storage class and are positioned outside functions. These items include functions and data items that are to be defined for reference from functions in other source files. Avoid using the dollar sign ($) as the eighth character of a function name since this may cause it to duplicate the name of a control section generated by the compiler for some other function.

You can define external objects with any name that does not begin with a dollar sign (**$**), two underscores (__) or an underscore followed by a letter. Certain library functions and data elements (defined in modules written in C) have names that start with an underscore (that appears as a pound sign (**#**) in the object code) or a dollar sign (**$**).

The extname option

To circumvent the above restrictions, the compiler enables you to specify the **extname** option, which permits extended names of up to 64K in length. An extended name is any name that identifies an **extern** variable or that identifies an **extern** or **static** function and fits either of the following criteria:

- is longer than eight characters
- is at most eight characters long, contains uppercase alphabetic characters, and is not the name of an **__asm** or high-level language (for example, **__pascal**) function.

External Variable Models

The compiler uses one of two reference-definition (ref/def) models for external variables, depending upon whether reentrant modification of external variables is allowed. The **rent** and **rentext** compiler options are used as follows to determine whether or not reentrant code is generated:

- If **norent** is specified, reentrant execution is not allowed and the compiler uses the strict ref/def model.
- If **rent** is specified, reentrant modification is allowed and the compiler uses the common ref/def model, unless the **refdef** compiler option is also specified. (The **refdef** option forces the use of the strict ref/def model.)

See “Reentrant and Non-reentrant Identifiers” on page 57 for more information about **rent** and **norent**.

Common ref/def model

The common model allows any number of definitions (including 0), with and without initializers, and any number of declarations of an external variable to be present. A single declaration is sufficient to create the variable. Only one definition of an external variable may specify an initializer. If more than one is encountered, the COOL linkage editor preprocessor issues a warning message, and the actual initial values used are unpredictable. If no initializations of a variable are present, then the variable is initialized to 0 as the C language definition requires.

Strict ref/def model

In contrast to the common model, the strict ref/def model requires exactly one definition, with or without an initializer, to be present for each external variable. Again, if no initializations of a variable are present, then the variable is initialized to 0.

Programming Considerations

Any program that conforms to the strict ref/def model also conforms to the common model. However, the reverse is not true. A program that takes advantage of the common model (for example, by omitting definitions for some variables) may not compile or link correctly when using the strict ref/def model. Also, a program that uses the common model and is compiled with the **rent** or **rentext** option may not work correctly when compiled with the **norent** option.

The ISO/ANSI C Standard mandates the strict ref/def model for external variables. If you plan to move a program to other machines, follow the strict ref/def model unless you are sure that all the other C compilers follow a more permissive model.

You should not link compilations that have been created with the **rent** and **rentext** options with compilations that have been created with the **norent** option. External variables cannot be shared between these two types of compilations, nor will the resulting load module be reentrant.

Object code produced by the compiler with the **rent** or **rentext** options must be preprocessed by COOL if external variables are initialized in more than one compilation. Conversely, object code produced by the compiler with the **norent** option does not need to be preprocessed by COOL. If you use the **rent** option rather than the **rentext** option, the initializations of static as well as external variables affect whether or not the use of COOL is required.

Reentrant and Non-reentrant Identifiers

This section describes reentrant and non-reentrant identifiers and explains how to use the **rent**, **rentext**, and **norent** compiler options as well as the **__rent** and **__norent** keywords to control whether an identifier is reentrant or non-reentrant.

The reentrant and non-reentrant attributes apply only to **extern** or **static** data. All functions, parameters, and automatic data are automatically reentrant.

Non-reentrant Identifiers

Non-reentrant data reside in a CSECT and are thus a part of the load module at execution time. If the program modifies non-reentrant data, it is modifying its own load module, thus rendering itself non-reentrant (that is, the same copy cannot be executed by multiple users or in multiple tasks simultaneously).

The compiler places non-reentrant data in the **static** CSECT. Wherever possible, non-reentrant data are initialized at compile time and are referenced using address constants (ACONs).

Reentrant Identifiers

Reentrant data are placed in the PRV (pseudoregister vector). The PRV is an area of memory allocated by the run-time library when the C program is executed. Thus, reentrant data can be freely modified without affecting the reentrancy of the program.

Reentrant data are referenced using Q-type address constants (QCONs). The QCON for a reentrant identifier contains the offset of the identifier from the start of the PRV. The offset is added to the address of the PRV to obtain the address of the data.

Placement of Data

In general, **static** and **extern** identifiers can be placed in either the non-reentrant section or the reentrant section (PRV). A number of rules control where a particular identifier is placed.

First, if you code an explicit **__rent** or **__norent** keyword, then the identifier is always placed in the section specified. With **__rent** and **__norent** you can control which section is used.

If you do not provide an explicit keyword, then generally the **rent**, **rentext** or **norent** compiler option determines the section into which the identifier is placed. **rent** places both **externs** and **statics** (without a **__rent** or **__norent** keyword) in the reentrant section. **norent** places them both in the non-reentrant section. **rentext** places **externs** in the reentrant section and **statics** in the non-reentrant section.

There are two exceptions (cases where the compiler option is overridden). The first exception occurs with **extern** identifiers whose names start with an underscore. These are always placed in the reentrant section. (The ISO/ANSI C Standard reserves such names for use by the compiler implementor).

The second exception occurs if the **const** keyword is used. **const** declares constant data. Since the data are not expected to change during program execution, the compiler tries to promote **const** identifiers into the non-reentrant section. The rules for this are fairly complicated. An identifier is promoted to the non-reentrant section if all of the following are true:

- The identifier is declared with the **const** keyword.
- The identifier is not declared with the **volatile** keyword.
- The identifier is not declared with the **__rent** keyword.
- The identifier has **static** scope or, if **extern** , the first letter of the identifier's name is not an underscore.
- The identifier is not a pointer, other than a local function pointer; and, if the identifier is an aggregate, it does not contain any pointers–including recursively generated pointers in any inner aggregates–other than local function pointers.

The **const** type qualifier is ignored (not used to place the data object in the string literal CSECT) in the following situations:

- if the identifier begins with an underscore (_)
- if the declaration also has the **volatile** type qualifier
- if the defined object is a pointer type or an aggregate that contains a pointer element, unless the pointer is a **__local** function pointer.

The last exception is a situation in which the pointer value cannot be determined until execution time; therefore, a value in the CSECT cannot be initialized without violating reentrancy.

Initialization and Reentrancy

If you plan for your program to be reentrant, you should not modify any data in the non-reentrant section when the program executes. You may also need to be careful when initializing non-reentrant data. Most initializations are done at compile time, which preserves reentrancy, but there are a couple of exceptions:

- initializing a pointer to the address of a reentrant identifier. The address of reentrant identifiers is not known until execution time since the PRV is allocated then.
- initializing a nonlocal function pointer. A nonlocal function pointer does not point directly to the function code but to additional information, some of which is not known until execution time.

To help you in such situations, the compiler produces a warning diagnostic if all of the following conditions are true:

- The initializers for an identifier in the non-reentrant section could require writing into the load module at execution time.
- The **rent** or **rentext** compiler option was specified; that is, you requested that a reentrant executable be created.

For a reentrant program, a simple way to avoid problems is to use only reentrant data. If your program is not intended to be reentrant, none of these considerations need concern you.

Declarations Must Agree

If you declare an identifier reentrant in one place and non-reentrant in another, you can end up with two different identifiers in the same compilation. Each of these identifiers will reference a location that is different from the other. The compiler will produce an error message in this situation.

This applies not only to explicit use of the **__rent** and **__norent** keywords, but to the implicit assignment of data to the reentrant and non-reentrant section using compiler options and the **const** keyword, as described above.

All declarations of an identifier must agree. Use the same **rent**, **rentext**, and **norent** settings for all compilations, or use the **__rent** or **__norent** keywords.

Cross-reference

The cross-reference tells you whether an identifier is reentrant (**rent** is listed) or non-reentrant (**norent** is listed). This information is produced regardless of how the reentrancy was determined (keywords, compilation options, etc). This may be helpful to you in cases where the rules above are not clear.

Sharing External Variables with Assembler Language Programs

External variables stored in the static CSECT can be accessed from assembler language programs via a V-type address constant (VCON). Accessing an external variable stored in a pseudoregister must be done indirectly by computing the offset of the variable in the pseudoregister vector. The address of the pseudoregister vector is stored at decimal offset 12 from the address in general register 12. (This area is known as CRABPRV.) Example Code 3.4 on page 59 sketches an assembler language routine that accesses an external variable stored as a pseudoregister.

Example Code 3.4 Sample Assembler Language Routine for Accessing an External Variable Stored as a Pseudoregister

```
LRX,12(,R12)     Set RX to point to CRABPRV. This
                      assumes that register 12 has the
```

```
                              CRAB address.
        ALRX,=Q(ZZZ)      RX now contains the address
                              of the external variable
        USING ZZZ,RX          ZZZ.  ZZZ is the name of the
                              variable as defined in the
                              C program.

        MVCZZZ1,=F'2'     Set integer variable to 2.
        .
        .
        .
ZZZ     DSECT                 Identify the dummy section ZZZ.
ZZZ1    DSF
```

Sharing External Variables with FORTRAN Programs

It is possible for a C program that has been compiled with **rent** or **rentext** to share data with FORTRAN programs. The values to be shared are in one or more FORTRAN COMMON blocks, where FORTRAN code can access them directly. C code accesses the COMMON blocks through function pointers. Each COMMON block is described by a C structure, with the structure tag the same as the name of the COMMON. Each COMMON block code is declared as a function of the same name. For example, the following code declares the COMMONs named **comona** and **comonb**:

```
extern comona();
extern comonb();
```

C code accesses the COMMONs through pointers. Each pointer is set up by invoking a macro, called **COMPTR**, in the example below. The complete definition for the **comona** structure is assumed to be elsewhere in the program:

```
#define comptr(block) (*(struct block**)&block)
.
.
.
struct comona *aptr;   /* Declare aptr as pointer to comona struct. */

aptr = comptr(comona);
aptr->field1 = 1;
.
.
.
```

Note that the structure tag, **comona**, is the same as the name of the COMMON. This is not required, but it simplifies the macro and makes the data sharing more obvious.

The technique is readily adaptable to other languages that implement shared data as CSECTs or COMMONs. However, this technique renders an otherwise reentrant program non-reentrant, by the nature of such implementations. See Chapter 16, "Implementing ILC with a User-Supported Language," in the {ilc} for more details.

Register Conventions and Patch Writing

The following sections describe register conventions and patch writing.

The Patch Area

By default, the compiler generates a patch area in each compiled module. This patch area provides space for you to apply maintenance to your modules in object code or load module form (zaps).

The patch area is generated in the first 4096 bytes of a CSECT known as the constants CSECT for the compilation. The constants CSECT is permanently addressed by general register 4, so the patch area is always addressable. The default patch area is 1/64 the size of the generated code for the compilation, rounded up to a multiple of 8 bytes, with a minimum size of 24 bytes and a maximum size of 256 bytes. The patch area is generated as a series of S-cons (address constants in base-displacement form). Each SCON contains its own address in base-displacement form, using register 4 as a base register. This minimizes errors in patch-writing, both for branches to the patch area and for branches within it.

To find the patch area for a module, look near the end of the OMD listing for the constants CSECT. (See "Compiler-generated Names" on page 53.) The patch area is found at the end of the CSECT or just before 4096 bytes if the CSECT exceeds 4096 bytes in size. It is easily recognized by its distinctive S-con format.

It is possible, for very large compilations, or when the zapspace option is specified, for more patch space to be required than the compiler can generate in the first 4096 bytes of the patch area. In this case, the compiler will generate one or more secondary patch areas later in the constants CSECT. These areas have the form of S-cons, using register 0 as the base register. Because these areas are not in the first 4K of the constants section, they are not directly addressable but can be branched to from the previous patch area.

The zapspace option

The **zapspace** compiler option can be used to alter the size of the compiler-generated patch area. The size of the patch area can be increased or its generation suppressed.

The **zapspace** option accepts an integer value between 0 and 22, inclusive, that specifies the factor by which the default patch area size is to be multiplied. If the factor is 0, then no patch area is generated. For example, if the default patch area is 48 bytes and the **zapspace** option specifies a factor of 3, then the patch area actually generated is 144 bytes long. In no case does the compiler generate more than 512 bytes of patch area.

The zapmin option

The **zapmin** compiler option can be used to specify the minimum size of the compiler-generated patch area.

The **zapmin** option accepts an integer value that specifies the number of bytes in the patch area. The default is 24. For example,

```
zapmin(64)
```

ensures that the patch area is at least 64 bytes.

In no case does the compiler generate more than 512 bytes of patch area.

Register Conventions

The following list summarizes register conventions. You need this information if you are writing patches.

- Register 4 (R4) addresses the constants CSECT, including the patch area. R4 always contains this address throughout execution.

- Register 5 (R5) is the base register for the current function.

  Unless the function exceeds 4K in size, R5 points to the start of the function. If the function exceeds 4K in size, examination of some branches near the place to be zapped allows the actual R5 value to be calculated.
- Register 12 (R12)addresses the CRAB that is required at execution time and can never be changed.
- Register 13 (R13) addresses the DSA (dynamic save area). The DSA includes automatic variables and the area for the outgoing parameter list. If the function uses more than 4095 bytes of auto variables, R13 directly addresses only the first 4095 bytes.
- Other registers are assigned usage dynamically based on need.

The best way to write a zap is to examine other code in the function that does something similar to what is required and to model the zap on that code.

General register 1 (R1) can be used as a scratch register for zaps unless it is already directly used in the zapped code or a function call sequence (which sets R1) is being zapped. The compiler never remembers the value in R1 across statements. If it is not possible to use R1, you should be aware that values can be kept in registers for a large number of statements, including conditional branches and function calls. The best register use for a zap (other than R1) is a register that is set soon after the zap without being used between the zap and the code that sets the register to a new value (that is, one in which the existing value is discarded).

CHAPTER

4

Optimization

The optimize Option

The **`optimize`** compiler option is used to enable global optimization, which optimizes the flow of control and data through an entire function.

Optimizations

Global optimization includes a wide variety of optimizations. Some of the optimizations that are performed are discussed below.

Register allocation

The compiler analyzes the function to determine which auto variables, formal variables, temporary values, and constant values should be assigned to registers at each point in the function. The compiler uses up to 6 of the 370 general registers and 2 of the floating point registers for this purpose. The remaining registers become either dedicated registers, as for example, in dedicating R12 as the CRAB pointer, or working registers during code sequences of medium or less duration.

Generally speaking, the variables that are most used at a given point are selected. Values occurring in loops are more likely to be chosen.

The compiler attempts to keep a variable assigned to a register for as long as possible. When 370 BXLE/BXH instructions are issued, the compiler allocates registers in pairs, resulting in high-quality code for many loops particularly in numerical applications.

Using the ampersand (&) operator with a variable prevents the compiler from allocating that variable to a register because it cannot predict when the resultant pointer will be used to read or modify the variable's value in memory, or the variable may be used in another function. External variables also cannot be allocated to a register.

The effect of global optimization's register allocation is quite different from the use of the **register** storage class. In general, a variable declared using **register** is associated with a machine register throughout the entire block in which it is declared (usually the entire function). In most functions, the variable is heavily used in some places and not used in others. Yet, if a machine register is assigned to the variable, then the same register cannot be reused even in those sections where the variable is not used. Therefore, global optimization changes a register's assigned variable during the evaluation of each expression to ensure that the most heavily used variables are always in machine registers.

The compiler overrides the register storage class keyword in the declarations of integer, double, and pointer variables.

Because of the portability of the C language, it would be difficult for a programmer to know the number of available registers provided by the target machine and the compiler. The concept of a register variable is based on the idea that the variable is kept in a register for the entirety of its scope. Such restrictions no longer apply when a compiler uses the more advanced registration allocation algorithms in SAS/C software. Even though the compiler does not have dynamic information about program execution that would indicate which statements are executed more heavily, it can use the loop nesting structure to make a reasonable approximation.

Dead store elimination

If a value is assigned to a variable, but the value is not used, then the assignment can be eliminated as in this example:

```
o = 23;

   code that does not refer to 'o'

o = 12;
```

The first assignment to **o** can be removed.

Since the compiler inspects all references to the variable throughout the entire function, even quite subtle dead stores are eliminated.

Moving invariant calculations out of loops

A calculation in a loop whose value is the same on each iteration can be moved outside of the loop. For example, the loop

```
for (i = 0; i < j; i++) {
   a[i] = p->q.r[10] ;
}
```

can be changed to

```
temp = p->q.r[10] ;
for (i = 0; i < j; i++) {
   a[i] = temp;
}
```

Refer to the explanation for the **`loop`** option in "Very busy expression hoisting" on page 66 for more information about this type of optimization.

Constant propagation and folding

References to a variable whose only definition is a constant are replaced by the constant. If the variable is used only in expressions with a different type (for example, if an **`int`** variable is only used in a comparison with **`float`** variables), global optimization creates a constant of the correct type. If the variable is used only as a constant, global optimization eliminates the variable entirely. The following example demonstrates these optimizations:

```
void f(double d)
{
   i = 10;
   for (; d < i; ++d) {
      .
      .
      .
   }
   return;
}
```

The code above can be changed to this:

```
void f(double d)
{
   for (; d < 10.0; ++d) {
      .
      .
      .
   }
   return;
}
```

Constant propagation is often useful in programs that contain inline functions.

Merging common subexpressions

Global optimization eliminates recalculation of values that have been computed previously. For example,

```
x = i / 3;
y = i / 3 + 4;
```

can be changed to

```
temp = i / 3;
x = temp;
y = temp + 4;
```

Dead code elimination

Code that can never be executed is eliminated.

Induction variable transformations

Loops containing multiplications (usually those associated with array indexing) have the operations changed to addition.

Very busy expression hoisting

If the same expression is computed along all paths from a point in the code, the expression is moved to a single, common location. For example,

```
if (expression )
   x = i + j;
else
   y = (i + j) * 2;
```

can be changed to

```
temp = i + j;
if (expression )
   x = temp;
else
   y = temp * 2;
```

Global Optimization Compiler Options

The compiler accepts the following options to modify optimization:

loop assumes that loops have multiple iterations when the number of iterations is variable. This enables the movement of safe code out of loops. (See “Moving invariant calculations out of loops” on page 65.) **loop** is the default.

When a loop is not executed at all, the moved code is executed in cases where it previously would not have been. For example,

```
for (i = 0; i < n; ++i)
   for (j = 0; j < m; ++j)
       p[i * m + j] += 1;
```

can be changed to

```
for (i = 0; i < n; ++i) {
   temp = i * m;
```

```
    for (j = 0; j < m; ++j)
       p[temp + j]  += 1;
}
```

In the changed code, **i*m** can be calculated when **m** is less than or equal to 0. When the **loop** option has been specified, there may be a small cost in time for every loop that is not executed. There is also a significant time saving for loops that are executed many times, as most are.

Some types of code may cause an exception, for example, division by 0.

For this reason, the SAS/C Compiler restricts moved code to safe operations, including integral and pointer arithmetic other than division by 0, but not including floating-point operations. The compiler avoids incorrect exceptions regardless of the setting of the **loop** option.

alias disables type-based aliasing assumptions. If **alias** is used, the compiler assumes worst-case aliasing. Use of this option can significantly reduce the amount of optimization that can be performed. **noalias** is the default.

greg controls the number of general registers that the compiler will try to allocate.

freg controls the number of floating-point registers that the compiler will try to allocate.

For both **greg** and **freg**, the compiler allocates from among the supported register variable registers. These are R6-R11 and FR4/FR6. Registers R4, R5, R12, and R13 are dedicated to addressing various data objects in the function. R1 is used for numerous specific code sequences.

R0, R2, R3, R14, R15, FR0, and FR2 remain for things like constants, base registers, VCONs, and nonregisterized variables. In the case that the user feels that values of these types are being reloaded too often from memory and can benefit from having more registers available, then the number of registers allocated with **greg** or **freg** can be reduced.

inline inlines small functions (as defined by the **complexity** option) and those with the **__inline** keyword. **inline** is the default when the **optimize** option is used.

inlocal inlines single-call static (local) functions.

complexity defines the complexity of functions considered small by **inline**. (See "Using the complexity option to control inlining" on page 70.)

depth defines the maximum depth of function calls to be inlined. The range is 0 to 6, and the default value is 3.

rdepth defines the maximum level of recursive function calls to be inlined. The range is 0 to 6, and the default is 0.

Global Optimization and the Debugger

The compiler does not optimize programs when the **debug** option is used. To utilize all the capabilities of the SAS/C Debugger, there must be an accurate correspondence

between object code and source line numbers, and optimizations can alter this correspondence. Also, the **debug** option causes the compiler to suppress allocation of variables to registers, so the resulting code is not completely optimal.

You can, however, use the **dbhook** option along with the **optimize** option to generate optimized object code that can be used with the debugger. The **dbhook** option generates hooks in the object code that enable the debugger to gain control of an executing program.

When using the debugger with optimized object code that has been compiled with the **dbhook** option, the source code is not displayed in the debugger's Source window and you cannot access variables. Therefore, the debugger's print command and other commands that are normally used with variables are not used when debugging optimized code. You can use commands such as step, goto, and runto to control the execution of your program. The goto command may cause incorrect results if the expected register contents at the goto target differ from the actual register contents when the command was issued. Also, source code line numbers are displayed in the Source window, providing an indication of your location in the code. You also have the capability of viewing register values in the debugger's Register window.

The debugging of optimized code is most effective when used in conjunction with the Object Module Disassembler (OMD) or your system's debugger. See Chapter 5, "Compiling C Programs," on page 81 for information about using the OMD.

The __inline Keyword for Inline Functions

An inline function is a function for which the compiler replaces a call to the function with the code for the function itself. The process of replacing a function call with the function's code is called inlining. When the compiler performs inlining for a function, the function has been inlined.

Overview

To define an inline function, add the **__inline** keyword to the function definition. The following is an example of a function definition using the **__inline** keyword:

```
__inline double square(double x)
{
   return x * x;
}
```

The **__inline** keyword causes a function to be inlined only if you specify the **optimize** option. If **optimize** is specified, whether or not **__inline** is honored depends on the setting of the **inline** optimizer option. By default, the **inline** option is in effect whenever the optimizer is run. If you specify **optimize**, you must also specify the **noinline** option if you want the **__inline** keyword to be ignored.

There are no restrictions on how an inline function can be coded. An inline function can declare auto variables and can call other functions, including other inline functions. Inline functions can also be recursive.

Advantages of Using Inline Functions

Since the call to an inline function is replaced with the function itself, the overhead of building a parameter list and calling the function is eliminated in the calling

function. Since there is no function call, the overhead associated with entering the function and returning to the caller is eliminated in the called function.

Below is an example of a program that calls an inline function. The program produces a table of equivalent temperatures using both the Fahrenheit and Celsius scales. The conversion from Fahrenheit to Celsius scale is done with the **ftoc** function.

```
#include <stdio.h>

static double ftoc(double);

void main()
{
   double fahr, celsius;
   puts("Fahrenheit  Celsius");
   for (fahr = 0.0; fahr <= 300.0; fahr += 20.0) {
      celsius = ftoc(fahr);
      printf("   %4.0f     %6.1f\n", fahr, celsius);
   }
}

static double ftoc(double fahr)
{
   return (5.0 / 9.0) * (fahr - 32.0);
}
```

As written, the program performs the following operations for each of the 16 iterations of the **for** loop:

1. builds a parameter list containing the value of **fahr**
2. calls the **ftoc** function
3. allocates stack storage for **ftoc**
4. calculates the temperature on the Celsius scale
5. stores the result of the calculation
6. frees the stack storage
7. returns to **main**
8. assigns the result to **celsius**.

Suppose **ftoc** is defined as an inline function by adding the **__inline** keyword, as follows:

```
__inline static double ftoc(double fahr)
{
   return (5.0 / 9.0) * (fahr - 32.0);
}
```

When the program is compiled using the **inline** option, the compiler replaces the call to **ftoc** with the code for the **ftoc** function, as shown here:

```
#include <stdio.h>

void main()
{
   double fahr, celsius;
   puts("Fahrenheit  Celsius");
   for (fahr = 0.0; fahr <= 300.0; fahr += 20.0) {
      celsius = (5.0 / 9.0) * (fahr - 32.0);
      printf("   %4.0f     %6.1f\ n", fahr, celsius);
```

```
    }
}
```

Note that the definition of **ftoc** has been moved to the main function. The static definition has been eliminated. Of the eight steps listed above, only two steps remain in the loop:

- calculate the temperature on the Celsius scale
- assign the result to **celsius**.

Disadvantages of Using Inline Functions

The compiler generates a copy of the code for an inline function at every call to the function. If the function is very large or is called in many different places, the size of the generated code for the program can increase dramatically. In addition, using inline functions may significantly increase the amount of time required to compile the program.

Compiler Options for Inlining

Several compiler options are supported to allow control over the amount of inlining performed by the compiler. These options are discussed in the following sections.

Using the inlocal option to control inlining

The **inlocal** option can be used to gain some of the benefits of inlining without using the **__inline** keyword. This option enables the inlining of all static functions that are called exactly once in the source program. By limiting inlining to single-call static functions, the **inlocal** option guarantees that the generated code for the program will not increase over the size when inlining is not used. In the preceding example, the same results can be obtained without using the **__inline** keyword by using the **inlocal** option when the program is compiled.

Using the complexity option to control inlining

The **complexity** option provides another way to use inlining without using the **__inline** keyword. If the **inline** option is in effect, then the compiler inlines small **static** and **extern** functions automatically even if they are not defined with the **__inline** keyword. The **complexity** option assigns a meaning to the word small and takes a value between 0 and 20, inclusive. For example, you may specify **complexity(4)** (**-Kcomplexity=4** under UNIX System Services [USS]). This specifies that the compiler should automatically inline all functions whose complexity is no higher than 4.

Complexity is a measure of the number of discrete operations defined by the function. In general, the larger the value specified for complexity, the larger the functions that are automatically inlined. The **ftoc** function, described earlier, has a complexity value of 1. The following function, which multiplies the two square matrices, **a** and **b**, and returns the result in matrix **c**, has a complexity value of 8:

```
void mmult(double c[] [10], double a[] [10] , double b[] [10] )
{
   int i, j, k;

   for (i = 0; i < 10; i++)
      for (j = 0; j < 10; j++) {
```

```
            c[i] [j]  = 0.0;
            for (k = 0; k < 10; k++)
               c[i] [j]  = c[i] [j] + a[i] [k] * b[k] [j] ;
         }
}
```

The following function, a simple binary search function, has a complexity value of 11. This example returns the index of the element in **list** that has the same value as **target**. **num_els** is the number of elements in the **list** array. **list** is sorted alphabetically. If **target** is not found, the function returns {minussym}1.

```
#include <string.h>

int binsrch(char *target, char *list[] , int num_els)
{
   int where, hit;
   int low, high, current;

   low = 0;
   high = num_els;
   current = num_els / 2;  /* Find middle element of array.     */
   hit = -1;               /* Target not found yet.             */

   do {
      where = strcmp(target,list[current] );
      if (where < 0)       /* Target is in top half of list.    */
         high = current - 1;
      else if (where > 0)  /* Target is in bottom half of list. */
         low = current + 1;
      else
         hit = current;                              /* success  */
      current = (high + low) / 2;
      } while (high >= low && hit < 0);

   return hit;
}
```

The optimizer default is **complexity(0)**, which means that no functions are considered small enough to inline unless they are defined with the **__inline** keyword. Note that using a high value for **complexity** can lead to a substantial increase in the size of the generated code for the compilation.

As mentioned earlier, inline functions can call other inline functions or call themselves recursively. You can control how the compiler generates code for sequences of calls to inline functions and for recursive inline functions by using the **depth** and **rdepth** options.

Using the depth option to control inlining

The **depth** option specifies a limit on the number of nested inline function calls. If inline function **f0** calls inline function **f1**, which calls inline function **fn**, then a single call to **f0** can result in a significant increase in the size of the function calling **f0**.

The following program shows how the compiler inlines functions that call other inline functions. This program computes the length of the hypotenuse of a triangle whose sides are of lengths **a** and **b**. The main function calls **hypot**, which in turn calls the **square** function.

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

static double hypot(double, double);
static double square(double);

void main()
{
   double a, b, c;

   for (a = 1.0; a < 10.0; a += 1.5) {
      b = a + 0.75;
      c = hypot(a, b);
      printf("a = %f, b = %f, c = %f\ n", a, b, c);
   }
}

static double hypot(double a, double b)
{
   return sqrt(square(a) + square(b));
}

static double square(double x)
{
   return x * x;
}
```

If both **hypot** and **square** are inline functions, then the compiler generates code for **main** as if the following program had been used.

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

void main()
{
   double a, b, c;

   for (a = 1.0; a < 10.0; a += 1.5) {
      b = a + 0.75;
      c = sqrt(a * a + b * b);
      printf("a = %f, b = %f, c = %f\ n", a, b, c);
   }
}
```

Note that the **square** function is inlined in **hypot**, which is then inlined in **main**. In this program, the maximum calling depth is 2.

If a long sequence of inline function calls is defined, then the size of the generated code for a compilation can increase greatly because of the number of functions being inlined. The **depth** option can be used to control the calling depth of inline functions. If the calling depth exceeds the number specified by the **depth** option, the compiler stops inlining and generates calls to the functions instead.

By default, the compiler uses a maximum calling depth of 3. The compiler accepts **depth** option values between 0 and 6, inclusive.

Using The rdepth option to control inlining

If the **rdepth** option is used, the compiler inlines recursive inline functions. The **rdepth** option specifies a maximum depth of recursive function calls to be inlined. The following program shows an example of this kind of inlining. The **fib** function calculates the Fibonacci function for its argument.

```
#include <stdio.h>

__inline static int fib(int);

void main()
{
   int i;

   for (i = 0; i < 10; i++)
      printf("fib(%d) = %d\ n", i, fib(i));
}

__inline static int fib(int i)
{
   if (i < 2)
      return i;
   else
      return fib(i-1) + fib(i-2);
}
```

If the program is compiled using **rdepth(2)**, then the compiler generates code as if the following program had been used:

```
#include <stdio.h>

static int fib(int);

void main()
{
   int i;
   int result1, result2, result;   /* compiler temporary variables */

   for (i = 0; i < 10; i++) {
      if (i < 2)
         result = i;
      else {
         if ((i - 1) < 2)
            result1 = i - 1;
         else
            result1 = fib((i - 1) - 1) + fib((i - 1) - 2);
         if ((i - 2) < 2)
            result2 = i - 2;
         else
            result2 = fib((i - 2) - 1) + fib((i - 2) - 2);
         result = result1 + result2;
      }
      printf("fib(%d) = %d\ n", i, result);
   }
}
```

```
static int fib(int i)
{
   if (i < 2)
      return i;
   else
      return fib(i-1) + fib(i-2);
}
```

The compiler has inlined code equivalent to the first two recursive calls to the **fib** function. This type of inlining can be very useful with recursive functions that have limited depth.

The maximum depth that can be specified using the **rdepth** option is 6. A large depth value can cause a large increase in the size of the generated code for the compilation. **rdepth(1)** is the default; that is, the compiler will not inline recursive functions.

The __actual Keyword for Inline Functions

There is no difference between **static** and **extern** functions defined using the **__inline** keyword. However, keep in mind that the compiler generally does not create a callable function for an inline function. This is not a problem if the function is declared **static** because all calls to the function are replaced with the inlined code for the function. However, **extern** inline functions are not callable from other compilations since no callable copy of the function exists.

The **__actual** keyword can be used in the definition of an inline function. **__actual** implies **__inline**, but it also specifies that the compiler should create a callable function as well. An **__actual extern** function is, of course, callable from other compilations just as any **extern** function.

Note that an extern **__inline** function in a source file generates no code, whether or not the optimizer is run, unless the **__actual** keyword is specified. This implies that if you have an extern **__inline** function declared in a header file, in order to support unoptimized or debug code, you must provide an **__actual** definition in some source file linked with the application.

This restriction does not apply to static **__inline** functions. When compiling without optimization, a static **__inline** function is compiled like any other static function. If you declare a static **__inline** function in a header file and compile without optimization, you will have a copy of its code in the object file for each source file that includes the header file. Thus, there is a tradeoff between declaring such functions static, which can waste memory on multiple copies of the code, and declaring them extern, which forces the programmer to add an **__actual** declaration for each such function to an appropriate source file. Which option is better will depend on the characteristics and requirements of the application.

Functions that Cannot Be Inlined

The compiler cannot inline a function

- that has its address taken
- that has a variable length argument list
- that is called with an argument list that does not agree with the declared parameter list.

Further Benefits of Inline Functions

There are additional benefits that occur when functions are inlined.

Extending the range of optimization

The value of using inline functions can go far beyond the obvious benefit of reducing function call overhead. In general, the compiler inlines the function and then optimizes the resulting code. Inlining often opens up additional possibilities for optimization. For example, if one or more arguments to an inline function are constant values, the compiler can often perform some of the computations at compile time.

Here is a simple example. Suppose the following program invokes the inline **ftoc** function given earlier:

```
#include <stdio.h>

void main()
{
   double fahr, celsius;

   fahr = 212.0;
   celsius = ftoc(fahr);
   printf("%fF is %fC\ n",fahr, celsius);
}
```

After inlining, the program looks like this:

```
#include <stdio.h>

void main()
{
   double fahr, celsius;

   fahr = 212.0;
   celsius = (5.0 / 9.0) * (fahr - 32.0);
   printf("%fF is %fC\ n",fahr, celsius);
}
```

Since the variables are assigned constant values, the compiler can compute the result of the calculation during compilation to produce code equivalent to the following program:

```
#include <stdio.h>

void main()
{
   printf("%fF is %fC\ n",212.0, 100.0);
}
```

Inline functions as replacements for macros

Since temporary auto variables can be defined in inline functions, often an inline function can be written that is easier to use than a macro.

Consider the problem of writing a function (called **strlength**) that has almost the same function as the Standard **strlen** function. The one difference is that, if the argument to **strlength** is NULL, then **strlength** returns 0. (The **strlen** function is not meaningful if called with a NULL argument.) A **STRLENGTH** macro is easily defined as follows:

```
#define strlength(p) ((p == NULL) ? 0 : strlen(p))
```

This macro works as described, but with one drawback. Its argument is evaluated twice, once in the test and once in the call to **strlen**. This is what is known as an

unsafe macro. If it is used with an argument that has side-effects, the result is usually incorrect. Suppose that the **STRLENGTH** macro is called as follows:

```
p = "A TYPICAL STRING";
n = strlength(p++);
```

The value assigned to **n** is 15, which is incorrect. (The intended result is 16.) This is because the macro expands to the statement shown here. (Note that **p** is incremented before being passed to **strlen**.)

```
n = ((p++ == NULL) ? 0 : strlen(p++));
```

However, it is easy to define an inline version of **strlength** that works correctly, as shown here:

```
__inline strlength(char *p)
{
   return (p == NULL) ? 0 : strlen(p);
}
```

Using inline functions to generate optimized code

As mentioned before, inlining often opens up additional possibilities for optimization. The following example shows how to use inline functions to take advantage of the compiler's capability to optimize the program after inlining is done.

The **pow** function, part of the C library, computes the value of **a** raised to a power **p** as expressed by the relation

```
r = a
```

[p] **pow** can be called with any real values for **a** and **p**. The following inline function, called **power**, supplants the **pow** function by generating inline code for constant, nonnegative whole number values of **p**. For **p**<=16.0, the compiler generates code to compute the value directly. For **p**>16.0, the compiler generates a loop to compute the result. If **p** is a variable, negative, or contains a nonzero fractional part, the compiler generates a call to the library **pow** function. In no case does the compiler generate code for more than one condition.

```
#include <lcdef.h> ❶ pos=m;
#include <math.h> ❷ pos=m;
#undef pow ❸ pos=m;
#define pow(x, p) power(x, p, isnumconst(p)) ❹ pos=m;

static __inline double power(double a, double p, int p_is_constant)
{

      /* Test the exponent to see if it's   */
      /*  - a compile-time integer constant */
      /*  - a whole number                  */
      /*  - nonnegative                     */
   if (p_is_constant && (int) p == p && (int) p >= 0) { ❺ pos=m;

      int n = p;

           /* Handle the cases for 0 <= n <= 4 directly. */
   ❻ pos=m;   if (n == 0) return 1.0;
         else if (n == 1) return a;
         else if (n == 2) return a * a;
         else if (n == 3) return a * a * a;
```

```
      else if (n == 4) return (a * a) * (a * a);

         /* Handle 5 <= n <= 16 by calling power            */
         /* recursively.                                    */
         /* Note that power is invoked directly, specifying */
         /* 1 as the value of the p_is_constant argument.   */
         /* This is because the isnumconst macro returns    */
         /* "false" for the expressions (n/2) and           */
         /* ((n+1)/2), which would defeat the optimization. */
❼ pos=m;   else if (n <= 16)
      return power(a, (double)(n/2), 1) *
                  power(a, (double)((n+1)/2), 1);

         /* Handle n > 16 via a loop.  The loop below       */
         /* calculates (a ** (2 ** x))                      */
         /* for 2 <= x <= n and sums the results for each   */
         /* power of 2 that has the corresponding bit set   */
         /* in n.                                           */
❽ pos=m;   else {
      double prod = 1.0;
      for (; n != 0; a *= a, n >>= 1)
         if (n & 1) prod *= a;
      return prod;
   }
}

   /* Finally, if p is negative or not a whole number, */
   /* call the library pow function.  The pow macro    */
   /* is defeated by surrounding the name "pow" with   */
   /* parentheses.                                     */
else ❾ pos=m;
   return (pow)(a, p);
}
```

The numbers in circles in the code above key the explanation that follows:

1 **<lcdef.h>** contains the definition of the **isnumconst** macro.

2 **<math.h>** contains the declaration of the **pow** function.

3 The **#undef pow** preprocessor directive undefines any macros that may be defined for the name **pow**.

4 This macro defines a **pow** macro that will cause the **power** inline function to be used instead of the library **pow** function. Note that the **isnumconst** macro is used to determine whether the second argument to **power** is a numeric constant. The result of **isnumconst** is passed as the third argument to **power**. (See Chapter 1, "Introduction to the SAS/C Library," in *SAS/C Library Reference, Volume 1* for more information about **isnumconst**.)

5 This is a constant expression and will be evaluated at compile time. The expression checks to determine if **p** is a numeric constant (as determined by **isnumconst**), a whole number, and greater than or equal to 0.

If an **if**-test is a constant expression (as is this one), the compiler evaluates the expression and then generates code only for the then branch or the else branch, depending on the result of the expression. In this example, if the result is false (that is, **p** is not a constant whole number greater than or equal to 0) then the compiler ignores the statements that compose the then branch and does not generate code to perform them. However, if the result of the expression is true,

then the compiler ignores the statements in the else branch and does not generate a call to the library's **pow** function.

6 This **if**-test, as well as the next four, are also constant expressions. As above, the compiler generates code for the return statement only if the result of the expression is true. Therefore, **if** 0≤**n**≤4, the compiler generates the appropriate return statement to compute the value of $\mathbf{a}^{\mathbf{n}}$ for **n**=1, 2, 3, or 4.

7 If 5≤**n**≤16, **power** is called recursively to evaluate $\mathbf{a}^{\mathbf{n}}$. Note that a program using this function should be compiled using the **rdepth** option with a recursion depth of 6.

8 For **n**>16, **power** uses a loop to compute $\mathbf{a}^{\mathbf{n}}$.

9 Finally, if **p** is nonconstant or is not a whole number, **power** calls the library's **pow** function to compute the result. Note that parentheses surround the name of the function. This defeats the macro definition for **pow** and ensures that a true function call is generated.

Here are some examples of the use of the power function:
Example 1

```
r = pow(a, 0);
```

Since the **if**-test (n== 0) is true, the compiler generates code to perform the statement

```
r = 1.0;
```

Example 2

```
r = pow(a, 2);
```

Since the **if**-test (n== 2) is true, the compiler generates code to perform the statement

```
r = a * a;
```

Example 3

```
r = pow(x, y);
```

Since **y** is not a constant, the compiler generates code to call the library's **pow** function:

```
r = (pow)(x, y);
```

Example 4

```
r = pow(x, 0.75);
```

Since **y** is not a whole number, the compiler again generates code to call the library's **pow** function:

```
r=(pow) (x,0.75);
```

Example 5

```
r = pow(x, 15.0);
```

Since 15.0> 4, **pow** calls itself recursively. The compiler generates code equivalent to

```
r = x * x * x * (x * x) * (x * x) *
    (x * x) * (x * x) * (x * x) * (x * x);
```

The computation above can be performed using only six floating-point multiplications. The following assembler language code illustrates the machine code instructions generated to compute $\mathbf{x}^{15}$:

```
LD    0,X      Floating-point register 0 (FPR0) = x.
LD    2,X      FPR2 = x, as well.
MDR   0,2      FPR0 = x * x = x
```

[2] MDR 2,0 FPR2 = x * x^2 = x^3 MDR 0,0 FPR0 = x^2 * x^2 = x^4 MDR 2,0 FPR2 = x^3 * x^4 = x^7 MDR 0,0 FPR0 = x^4 * x^4 = x^8 MDR 2,0 FPR2 = x^8 * x^7 = x^{15}

Note that programs that use the **power** function as shown should be compiled using these options: **optimize**, **rdepth 6**, **depth 3**. Since the function is defined using the **__inline** keyword, the **complexity** option is not required. If the **__inline** keyword is not used, you need to specify the **complexity** option with a value of at least 16.

Efficient Programming with the SAS/C Compiler

This section suggests several ways to write more efficient C code. By taking advantage of compiler features and the way the compiler generates code for certain C constructs, you can write programs that execute faster and more efficiently.

Using Leaf Functions

A leaf function is a function that calls no other functions. This property means that the function is always at the end of a calling sequence. The compiler can tell that a leaf function calls no other functions and takes advantage of this information. Instead of needing a fresh allocation of stack space, a leaf function uses a fixed area of storage in the CRAB for its automatic variables (as long as they do not exceed 128 bytes). This leads to a much more efficient entry and return sequence for these functions. You should make heavily used functions leaf functions where possible. Leaf functions are also known as DSA-less functions because no DSA is required.

Taking Advantage of Switch Optimizations

The compiler chooses from several possible algorithms when generating code for a switch statement. In some instances, the compiler can generate code for switch statements by using indexed tables with 1- or 2-byte entries. This ensures quick execution and also minimizes the amount of dataspace used.

For every switch encountered in the source file, the code generator analyzes the size and execution time that would result from each algorithm and chooses the best one. For switches with a small number of cases, one of the nonindexed methods is generally used since the overhead of the table lookup is not justified. However, for a large number of cases, an indexed algorithm is used for all but highly sparse switch statements.

Indexed algorithms require one table entry for each value in the switch range (the difference between the lowest and highest values in case statements). Therefore, it is advantageous to reduce the range of switch statements if possible because this reduces table space. If you have one or two case values that are very different from the others, you may want to test for them separately or handle them at the default label (which is not part of the range).

Optimization and Far Pointers

Most programs that use far pointers access a small number of address spaces. A typical design may use only the primary address space and one or two dataspaces. Because the optimizer and compiler cannot in general tell whether two far pointers

reference the same address space, there may be a lot of unnecessary reloading of the access registers.

A technique that may lead to better generated code is to use only a few far pointers (one per secondary address space or dataspace) and then to use offsets rather than pointers to address objects in the dataspaces. For instance, in place of the following code:

```
__far struct cb *lookup_user(char *);
__far struct cb *userptr;

userptr = lookup_user("fred");
userptr->inuse = 1;
```

You could use the following equivalent code instead:

```
extern __far char *user_data_space;
int lookup_user(char *);
int useroff;

useroff = lookup_user("fred");
((__far struct cb *)(user_data_space + useroff))->inuse = 1;
```

This style allows the optimizer to recognize that the variable **user_data_space** is frequently accessed, and should have a register reserved for it. This may in turn mean that the ALET for the user dataspace will only need to be loaded from memory once during the execution of a particular function.

This style of coding can be made more readable by using the preprocessor, as shown in the following example:

```
#define contents(type, alet, offset) \
        ((__far type *)((alet) + (offset)))
#define offset(alet, addr) \
        (((__far char *) addr) - alet)
#define cb_contents(offset) \
        contents(struct cb, user_data_space, offset)

extern __far char *user_data_space;
int lookup_user(char *);
int useroff;

useroff = lookup_user("fred");
cb_contents(useroff)->inuse = 1;
```

CHAPTER 5

Compiling C Programs

Introduction

The following sections describe how to compile C programs in TSO, under CMS, from the UNIX System Services (USS) shell, and under OS/390 batch.

Compiling C Programs in TSO

This section explains how to use the LC370 CLIST, which invokes the SAS/C Compiler. Included are discussions on specifying data sets and compiler options in TSO. Since the compiler 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 before you use the compiler. Your installation will probably cause it to be allocated automatically. Consult your SAS Installation Representative for C compiler products to determine if this has been done. If not, use the TSO ALLOCATE command to associate the library with the DDname CTRANS.

The LC370 CLIST

Invoke the compiler with the LC370 CLIST as follows:

```
LC370 dsname  [options]
```

The *dsname* argument is the name of the data set containing the source to be compiled. The *options* arguments are compiler options (see Chapter 6, "Compiler Options," on page 101). For the data set specification, you must follow standard TSO naming conventions; that is, if the data set belongs to another user, you must specify the full name of the data set and enclose the full name in single quotes. If the source code is in a member of a partitioned data set, you must specify the member name in parentheses following the data set name, in the normal TSO manner. For example, a data set belonging to another user can be specified as follows:

```
'YOURLOG.PROJ4.C'
```

If a member name is included, the data set can be specified as follows:

```
'YOURLOG.PROJ4.C(PART1)'
```

If you do not enclose the data set name in single quotes, the LC370 CLIST assumes that the final qualifier of the data set name is C. If you do not specify C, it is added automatically by the CLIST. (The default qualifier can be changed by your site when the CLIST is installed.) For example, in the command

```
LC370 PROJ4(PART1)
```

the C source is assumed to be in member PART1 of the data set *prefix*.PROJ4.C, where *prefix* is the user's default prefix. You must separate options with one or more commas, blanks, or tabs. The only order requirement is that the input source filename must be the first item on the command line following LC370. You can abbreviate options to the portion shown in uppercase in Table 6.1 on page 102. To negate an option, precede it with **NO**. If you use both the positive and negative form of an option (for example, **debug** and **nodebug**), both options are rejected and the default is in effect.

Sample LC370 CLIST command line with options

The following is a sample command line that invokes the compiler and executes the OMD:

```
LC370 PROJ4(PART1) PR('''ADMIN.PROJ4.LIST''') COMN SO NOX OMD V
```

The following items discuss the sample command line:

- The command line invokes the CLIST to compile member PART1 of data set *prefix*.PROJ4.C, where *prefix* represents the TSO prefix for the user. (The prefix is usually the same as the user's ID.)

- Since neither **object** nor **noobject** is specified and the source data set is not enclosed in single quotes, **object** is the default. Object code is stored by default in the data set *userid*.PROJ4.OBJ, member PART1.
- The **print** option specifies that the listing file should be generated and stored in data set ADMIN.PROJ4.LIST. This data set belongs to another user and is therefore set off by three single quotation marks before and after the data set name.
- The **lib** option is not present; the default **nolib** is therefore in effect. No header file libraries are needed other than the standard include library.
- The compiler options specified are **comnest**, **source**, **noxref**, and **omd**. The **verbose** option for the OMD is also specified.

For information on the OMD, see "The Object Module Disassembler" on page 95.

Compiling C Programs from the USS Shell

This section explains how to invoke the SAS/C Compiler from the MVS/ESA USS shell. As explained in Chapter 15, "Developing Applications for Use with UNIX System Services OS/390," on page 331, the shell provides an operating system interface with commands and features that are very similar to a UNIX operating system.

The following syntax is used to compile and/or link a program from the USS shell:

```
sascc370 [options] [filename1 [filename2 ... ]]
```

The *options* argument is a list of compiler options (see Chapter 6, "Compiler Options," on page 101), COOL options and/or OS/390 linkage editor options (see Chapter 7, "Linking C Programs," on page 131). The *filename* arguments may contain any combination of C source files, object modules, and AR370 archives. Any input source files are compiled and then linked with the object files and the archives.

sascc370 invokes the compiler if there are any input sources files and then invokes the COOL processor followed by the linkage editor to link the object files. The COOL and link-edit steps will be bypassed if you specify the **-c** option. Note that this behavior is different from the TSO and CMS behavior, where separate commands are required to compile and link.

The input files to be compiled may reside either in the USS hierarchical file system (HFS) or in a standard OS/390 partitioned data set. The use of an HFS file is illustrated in the following example:

```
sascc370 proj/sort.c
```

If the program was located in the OS/390 PDS member named YOURLOG.PROJ4.C(SORT), the compiler could be invoked from the USS shell as follows:

```
sascc370 '//dsn:yourlog.proj4.c(sort)'
```

Alternately, assuming the initial qualifier YOURLOG is also your userid, the command could be shortened to:

```
sascc370 '//proj4.c(sort)'
```

In either case, the compiled and linked output module is stored in the file **a.out** in your current directory. To specify another file, use the **-o** option. For instance, the following command stores the output module in the file **./proj5/sort**.

```
sascc370 -o ./proj5/sort ./proj5/sort.c
```

Here is an example of compiling a source program and linking it into an OS/390 PDS. The compiler options **-Kcomnest** and **-Krent** are used. Also, the linkage editor option **-Brent** is specified:

```
sascc370 -Kcomnest -Krent -Brent -o '//proj5.load(sort)' ./proj5/sort.c
```

Note that in order to invoke the **sascc370** command, you must include the directory where SAS/C was installed in your PATH environment variable. Probably, your site will define PATH appropriately for you when you start up the shell. If your site does not do this, contact your SAS Installation Representative for C compiler products to obtain the correct directory name and add it to your PATH.

Compiling C Programs under CMS

This section explains how to use the LC370 EXEC, which invokes the SAS/C Compiler under CMS.

The LC370 EXEC

Invoke the compiler with the LC370 EXEC as follows:

```
LC370 filename [[.] filetype [[.] filemode]] [(options [)]]
```

or

```
LC370 ddn:ddname  [(options [)]]
```

where *filename* is the name of a C source file. The default *filetype* is C. If *filemode* is not specified, all accessed disks are searched. A DDN: type filename is illustrated in the following example:

```
LC370 DDN:SYSIN
```

where SYSIN is interpreted as a DDname defined by a FILEDEF. The following example is also acceptable:

```
LC370 DDN:ddname (member)
```

where *member*, which must be enclosed in parentheses, refers to a member of an OS/390 PDS. The *member* must immediately follow the DDname.

You can also specify Shared File System (SFS) files as input and output when you invoke the compiler. Specify SFS files when invoking the compiler as follows:

```
LC370 sf:filename  [filetype  [dirname]]
```

where *dirname* is the complete directory name or the NAMEDEF that has been logically assigned to it. If you omit *filetype* or *dirname*, the default filetype is C; the default directory name is a period (.), indicating the top directory.

The compiler writes its output files (LISTING, TEXT, and temporary) to different places, depending on the form of the input fileid. With the CMS fileid, the compiler writes output files to the input file minidisk, if that minidisk can be written to. Otherwise, the compiler writes output files to the A-disk. With the DDN: format, the compiler uses the DDname as the filename of the output files (with an appropriate filetype). The output files are written to the input file minidisk if possible. Otherwise, they are written to the A-disk. With the SFS fileid, the compiler writes the output files to the input file directory if that directory is writable. Otherwise, the compiler writes output files to the top directory.

You should issue a GLOBAL command for any MACLIB containing **#include** files before invoking LC370. The standard C macro library is in LC370 MACLIB. You also can set default options and default MACLIBs through a GLOBALV variable. See "Using Environment Variables to Specify Defaults" on page 85.

Sample LC370 EXEC command line with options

The following is another sample command line that invokes the compiler:

```
LC370 PROG (COMN PR TRA NOX
```

In this command, the fileid of the source file is PROG C. The compiler options used are **comnest**, **print**, **trans**, and **noxref**. These options are discussed in Chapter 6, "Compiler Options," on page 101.

Note: The LC370 EXEC does not accept the short form of the compiler options. △

Compiling C Programs from XEDIT

LCXED is an XEDIT macro that invokes the compiler from within XEDIT. When you submit the LCXED command, the file currently being edited is compiled. To use the LCXED macro, enter the following on the XEDIT command line:

```
LCXED [(options  [)]]
```

options can be any options acceptable to the LC370 EXEC.

Using Environment Variables to Specify Defaults

The compiler references certain environment variables

- to set default values for LC370 EXEC and LCXED XEDIT options
- to specify a list of directories to be searched for included files
- to create a default list of MACLIBs that need to be GLOBALed when the compiler is invoked
- to create a default list of TXTLIBs that need to be GLOBALed when COOL is invoked.

The compiler and COOL query environment variables in the GLOBALV group LC370 to determine if any default options, MACLIBs, or TXTLIBs have been specified. Table 5.1 on page 85 lists the environment variables that can be defined in the LC370 group.

Table 5.1 GLOBALV Group LC370 Variables

Variable	Contents
_DB	directory list to search for debugger table file
_HEADERS	list of directories to search for included files
_INCLUDE	list of files in CLINK INCLUDE statement
MACLIBS	MACLIB(s) to be GLOBALed when the compiler is invoked

Variable	Contents
OPTIONS	compiler options
TXTLIBS	TXTLIB(s) to be GLOBALed when CLINK is invoked

Specifying Shared File System Directories

If you are using the CMS Shared File System (VM/SP Release 6 and later), you can use the _HEADERS and _INCLUDE environment variables to indicate a directory list for the compiler to search.

- The _HEADERS environment variable specifies the directory list to search for files included with the **`#include`** preprocessor directive.
- The _INCLUDE environment variable specifies the directory list to search for TEXT files included with the COOL INCLUDE control statement.

The syntax of specifying an environment variable is as follows:

```
GLOBALV SELECT LC370 SETL environment-variable directory-list
```

where *directory-list* is the list of directories that you want to be searched. You may specify either a directory name or a NAMEDEF when listing a directory that you want to be searched. The directories specified by the environment variable are searched in the order in which you listed them.

For example,

```
GLOBALV SELECT LC370 SET _INCLUDE .C.PROJ1 .C.PROJ2
```

instructs the compiler to search the .C.PROJ1 directory first and the .C.PROJ2 directory second as it looks for TEXT files.

You can also specify additional directory lists by defining other environment variables in the LC370 group. In this way, you can expand the list of directory names or NAMEDEFs that you want the compiler to search. An example of defining an environment variable for this purpose is as follows:

```
GLOBALV SELECT LC370 SETL MORE .SYSTEM.H .LOCAL.H
```

This example defines MORE as an environment variable that contains the .SYSTEM.H and .LOCAL.H directory names. Then, for example, you can specify the MORE environment variable within the _HEADERS variable by preceding it with an ampersand (&), as in the following:

```
GLOBALV SELECT LC370 SETL _HEADERS .PROJECT.H .COMMON.H &MORE
```

In this example, the compiler searches the .PROJECT.H and .COMMON.H directories and the .SYSTEM.H and .LOCAL.H directories listed in the MORE environment variable.

You can mix directory names, NAMEDEFs, and environment variables within an environment variable. Secondary environment variables, such as MORE in the preceding example, can specify tertiary environment variable names, and so on, to any depth. An example of specifying a tertiary environment variable follows:

```
GLOBALV SELECT LC370 SETL MORE .SYSTEM.H .LOCAL.H &MORE2
```

In this example, the environment variable MORE lists the .SYSTEM.H and .LOCAL.H directories and the directories listed in the &MORE2 environment variable. Note again that the tertiary environment variable defined in MORE must be preceded by an ampersand (&). The ability to list environment variables in successive levels circumvents the environment variable limit of 255 characters.

If the compiler does not find the included file in any directory that an environment variable specifies, it behaves as though you have not specified an environment variable at all. The compiler searches the top directory first. If it does not find the included file in the top directory, the compiler searches the accessed minidisks.

Other Environment Variables

You can specify a list of default compiler options with the OPTIONS environment variable. If compiler options are specified both by using the OPTIONS variable and on the command line, the command-line options override those specified by the OPTIONS variable. You can specify a default list of macros to be searched for header files with the MACLIBS environment variable. You can specify a default list of TXTLIBs to be used for autocall in COOL with the TXTLIBS environment variable. The EXECs also retain the current GLOBALed MACLIBs and TXTLIBs before issuing a new GLOBAL command, and they restore the status after the compiler or COOL has terminated. All of the EXECs accept the **`noglobal`** option. If this option is used, no MACLIBs or TXTLIBs are GLOBALed by the EXEC. For example, the following CMS GLOBALV command creates a list of default options to be used when the compiler is invoked:

```
GLOBALV SELECT LC370 SETL OPTIONS COMNEST RENT HLIST
```

The following GLOBALV command specifies that the LC370 MACLIB is to be GLOBALed when the compiler is invoked:

```
GLOBALV SELECT LC370 SETP MACLIBS LC370
```

Compiling C Programs under OS/390 Batch

This section discusses the cataloged procedures and compiler JCL requirements necessary for executing the compiler under OS/390 batch.

The LC370C Cataloged Procedure

The procedure LC370C runs the compiler and, optionally, runs the object module disassembler. The object module disassembler can be used as an aid in debugging at the machine-code level. If you want to execute the OMD by itself, refer to “The LC370D cataloged procedure” on page 98. See Example Code 5.1 on page 87 for typical JCL to run LC370C.

Example Code 5.1 Sample JCL for Compiling with Procedure LC370C

```
//COMPILE  JOB job card information
//*---------------------------------------------------------------
//*   COMPILE A C PROGRAM
//*   REPLACE GENERIC NAMES AS APPROPRIATE
//*---------------------------------------------------------------
//STEP1     EXEC  LC370C,PARM.C='options'
//C.SYSLIN   DD  DISP=OLD,DSN=your.object.library(member)
//C.SYSIN    DD  DISP=SHR,DSN=your.source.library(member)
//C.libddn   DD  DISP=SHR,DSN=your.macro.library
//
```

When you use LC370C, you only need to provide DD cards for SYSIN (your source data set) and SYSLIN (your object data set). Use the DD statement C.*libddn* to identify the macro library for any **#include** '*member.libddn*' statements in your source code. Refer to "Include-File Processing" on page 11 for detailed information on **#include** files.

The LC370C procedure contains the JCL shown in Example Code 5.2 on page 88. This JCL is correct as of the publication of this guide. However, it may be subject to change.

Example Code 5.2 Expanded JCL for LC370C

```
//LC370C PROC
//*************************************************************
//*  PRODUCT:  SAS/C                                          ***
//*  PROCEDURE:  COMPILATION                                  ***
//*  DOCUMENTATION:  SAS/C COMPILER AND LIBRARY USER'S GUIDE   ***
//*  FROM:  SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC     ***
//*************************************************************
//*
//C        EXEC PGM=LC370B
//STEPLIB  DD DSN=SASC.LOAD,
//            DISP=SHR COMPILER LIBRARY
//         DD DSN=SASC.LINKLIB
//            DISP=SHR RUNTIME LIBRARY
//SYSTERM  DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
//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=&MACLIB,
//            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
/*
```

Note the following about this JCL:

- When you override SYSPRINT to reference a disk data set, the data set disposition must be MOD. The data set must not be a member of a PDS.
- SYSLIN and SYSIN may address files in the Hierarchical File System.
- *libddn* statements may address an HFS directory containing user include files.

Sample use of LC370C with options

The following is a sample EXEC statement that invokes LC370C and passes compiler options via the PARM parameter:

```
// EXEC LC370C,PARM.C='COMNEST,NOTRANS,PRINT,NOXREF'
```

Compiler Options (Short Forms)

The SAS/C compile process is divided into several phases. Calls to each phase are normally controlled by a front-end command processor. These front-end processors

accept what are referred to as long-form options. When invoking the various phases, the front-end processors convert the options applicable to each phase to a form referred to as short-form options. Each phase only accepts the short-form versions of its options.

If you write your own JCL to invoke the compiler phases, you must pass short-form options in place of the options documented elsewhere in this book. You should not pass short-form options to any of the standard front ends, as the results are unpredictable.

The option string is case insensitive and has the following format:

```
c: compiler-options  p: listing-options  x: xref-options
    o: optimizer-options
```

where *compiler-options*, *listing-options*, *xref-options*, and *optimizer-options* are any or none of the options listed in Table 5.2 on page 90. Options must be separated by blanks. You can suppress the production of a source listing by using **!p:** instead of **p:**. Similarly, you can suppress the cross-reference by using **!x:** instead of **x:**. (Note that the not sign (¬) can be used instead of the exclamation point (**!**). This has the advantage that ¬ prints in your JCL listing even on a printer without the full C character set.) To suppress global optimization, you omit the LCGO step and specify the quad file produced by phase 1 as input to LC2370. Following are two examples of an options string.

The first example is as follows:

```
// EXEC PGM=LC1370,PARM='p: !t -i -h !x:'
```

This is equivalent to the following LC370C specification to the LC370B front end:

```
// EXEC LC370C,PARM.C='NOTRANS,ILIST,HLIST,NOXREF'
```

The second example uses uppercase characters and the ¬ instead of **!**":

```
// EXEC PGM=LC1370,PARM='C: -RX -SXYZ ¬P: O: -L'
```

This parameter string is equivalent to the following:

```
// EXEC LC370C,PARM.C='RENTEXT,SNAME(XYZ),NOPRINT,OPTIMIZE,LOOP'
```

Options for the global optimization phase should be supplied both to phase 1 and to the global optimizer. When passing them to phase 1, they should be preceded with **o:**. For example, suppose the following EXEC statement is issued:

```
// EXEC PGM=LC1370,PARM='o: -il !l'
```

This is equivalent to the following LC370C specification to the LC370B front end:

```
// EXEC LC370C,PARM.C='OPTIMIZE,INLOCAL,NOLOOP'
```

However, omit the **o:** when you pass the options to the global optimization phase. Remember that phase 1 just prints the global optimization options on the listing. You must pass them as parms to the global optimization phase for them to take effect.

Table 5.2 on page 90 lists the short-form equivalents for the compiler, listing, cross-reference, and optimizer options. In general, each option is one or two characters. Precede the short form of a compiler option with a hyphen (–) for the positive form and with an exclamation point (!) or not sign (¬) for the negative form. (The negative form is the equivalent of a long form with the **NO** prefix.) For simplicity, only the positive form of each option is shown, even if the default value is negative. For example, **-r** is given for **rent**, even though the default value is **norent**. See Chapter 6, "Compiler Options," on page 101 for a complete explanation of each option.

These compiler long form and short form options are functionally equivalent, but they are not implemented identically. Compiler long form options can be used only when invoking the compiler with the LC370B driver, which is called by the standard SAS/C cataloged procedures. LC370B converts the long form options to short form options for subsequent processing by LC1370.

Compiler short form options should be used only when directly calling LC1370. Passing short form options in the PARM string when using a SAS/C cataloged procedure produces unpredictable results.

Table 5.2 Compiler Option Equivalents

Long Form	Short Form	Long Form	Short Form
Compiler			
armode	**-ea**	**asciiout**	**-ao**
at	**-ca**	**bitfield(*n*)**	**-bf*n***
bytealign	**-b**	**comnest**	**-cc**
cxx	**-cxx**	**dbgmacro**	**-xp**
dbhook	**-xlo**	**debug**	**-d**
define(*sym=val*)	**-d*sym=val***	**dollars**	**-cd**
enforce(*n*)	**-y—*n***	**extname**	**-n!**
files(*xxx*)	**-f*xxx***	**hmulti**	**-ih**
igline	**-igl**	**indep**	**-i**
ipath	**-i$**	**japan**	**-j**
knobinder	**none**	**lineno**	**-l**
mention(*n*)	**-y+*n***	**pflocal**	**-4**
posix	**-lp**	**ppix**	**-co**
pponly	**-p**	**redef**	**-cr**
refdef	**-rd**	**rent**	**-r**
rentext	**-rx**	**reqproto**	**-cf**
sname(*name*)	**-s*name***	**stmap**	**-s**
strict	**-ll**	**stringdup**	**-cs**
suppress(*n*)	**-y*n***	**term**	**-t**
trigraphs	**-cg**	**undef**	**-u**
usearch	**-hu**	**vstring**	**-v**
zapmin(*n*)	**-zm*n***	**zapspace(*n*)**	**-z*n***
Listing			
enxref	**-n**	**exclude**	**-e**
hlist	**-h**	**ilist**	**-i**
maclist/mlist	**-m**	**options**	**-o**
overstrike	**-to**	**pagesize(*nn*)**	**-p*nn***
print	**:p**	**source**	**-s**
trans	**-t**	**upper**	**-u**
warn	**-w**		
Cross-Reference			

Long Form	Short Form	Long Form	Short Form
hxref	**-h**	**ixref**	**-i**
xref	**x:**		
Optimizer			
alias	**-a**	**complexity(*n*)**	**-ic*n***
depth(*n*)	**-id*n***	**freg(*n*)**	**-rf*n***
greg(*n*)	**-rg*n***	**inline**	**-i*n***
inlocal	**-il**	**loop**	**-l**
optimize	**o:**	**rdepth(*n*)**	**-ir*n***

Compiler JCL Requirements

This section discusses the data definition statements needed to run the compiler under OS/390 if you are writing your own JCL. The short forms of the compiler options, used when invoking the compiler without using the LC370B front end, are given. The compiler runs in three phases. The first phase, LC1370, reads the source file and produces an intermediate file (called the quad file) and the optional source listing. The second phase, LCGO, reads the quad file and produces a new, optimized quad file. LCGO is the global optimization phase referred to elsewhere in this manual. This phase is optional (unless you are compiling with **debug**, in which case it cannot be used because **debug** and **optimize** are not compatible). If LCGO is not used, the quad file produced by LC1370 can be input directly to the third phase. The third phase, LC2370, reads the quad file and generates the object module.

All compiler options are processed by LC1370, with the exception of the global optimization options. These are processed by LCGO but are also accepted by LC1370 so that they can be printed on the listing. If you are writing your own JCL, be sure that you supply the same global optimization options to both LC1370 and LCGO to ensure that the options used by LCGO are the same as the options printed by LC1370.

In summary, you can choose to run the compiler with or without the global optimization phase. If you run the compiler without the global optimization phase, you can also use the **debug** option. Sample JCL for each of these approaches is provided following Table 5.3 on page 91. You need the data definition (DD) statements shown in Table 5.3 on page 91 to invoke the compiler.

Note: All of the DDnames shown in Table 5.3 on page 91 can be specified as either an OS/390 data set or an HFS filename, with the exceptions of SYSLIB, H, SYSTERM, and SYSDBLIB. SYSLIB and H can be specified as either an OS/390 PDS or an HFS directory, whereas, SYSTERM and SYSDBLIB must not be HFS files or directories. △

Table 5.3 Data Definition Statements for Program Compilation

DDname	Purpose
LC1370	
SYSLIB	**#include** files (implementation-provided)
H (or other)	**#include** files (user-provided)
SYSPRINT	compiler listing and error messages

DDname	Purpose
SYSTERM	run-time error messages
SYSUT1	intermediate file (passed to LCGO or LC2370)
SYSUT2	debugger information file (only needed if **debug** in effect)
SYSIN	input source (sequential or PDS member)
LCGO	
SYSPRINT	global optimization messages
SYSTERM	run-time error messages
SYSUT1	intermediate file from phase 1
SYSUT3	optimized intermediate file (passed to LC2370)
LC2370	
SYSPRINT	compiler messages
SYSTERM	run-time error messages
SYSUT1	intermediate file from LCGO or phase 1
SYSUT2	debugger information file from phase 1 (only if **debug** is in effect)
SYSDBLIB	output debugger file (must be a partitioned data set)
SYSLIN	output object module (sequential or PDS member)

All files other than SYSTERM can be USS HFS files.

Running the compiler with the global optimization phase

The sample JCL in Example Code 5.3 on page 92 illustrates how to run the compiler under OS/390 while including the global optimization phase.

Example Code 5.3 Sample JCL for Running the Compiler under OS/390 with the Global Optimization Phase

```
//COMPILE    JOB  job card information
//*-------------------------------------------------------------
//*   EXAMPLE JCL FOR COMPILATION.
//*   REPLACE GENERIC NAMES AS APPROPRIATE.
//*-------------------------------------------------------------
//*   SYNTAX ANALYSIS PHASE
//*-------------------------------------------------------------
//STEP1     EXEC  PGM=LC1370,REGION=1024K,
//    PARM='-R P: -M !X: O: -IN -IL -IC8'
//STEPLIB     DD  DISP=SHR,DSN=compiler.library
//            DD  DISP=SHR,DSN=runtime.library
//SYSLIB      DD  DISP=SHR,DSN=standard.macro.library
//H           DD  DISP=SHR,DSN=your.macro.library
//SYSPRINT    DD  SYSOUT=class
//SYSTERM     DD  SYSOUT=class
//SYSUT1      DD  UNIT=SYSDA,SPACE=(CYL,(1,1)),
//    DISP=(NEW,PASS),
//    DSN=&&QUADS
//SYSIN       DD  DISP=SHR,DSN=your.source.library(member)
//*
```

```
/*
//*-------------------------------------------------------------
//*   GLOBAL OPTIMIZATION PHASE
//*-------------------------------------------------------------
//STEP2     EXEC  PGM=LCGO,REGION=2048K,COND=(4,LT),
//    PARM='-IN -IL -IC8'
//STEPLIB     DD  DISP=SHR,DSN=compiler.library
//            DD  DISP=SHR,DSN=runtime.library
//SYSPRINT    DD  SYSOUT=class
//SYSTERM     DD  SYSOUT=class
//SYSUT1      DD  DSN=&&QUADS,DISP=(OLD,PASS)
//SYSUT3      DD  UNIT=SYSDA,SPACE=(CYL,(1,1)),
//    DISP=(NEW,PASS),DSN=&&NEWQ
//*
/*
//*-------------------------------------------------------------
//*   CODE GENERATION PHASE
//*-------------------------------------------------------------
//STEP3     EXEC  PGM=LC2370,REGION=1024K,COND=(4,LT)
//STEPLIB     DD  DISP=SHR,DSN=compiler.library
//            DD  DISP=SHR,DSN=runtime.library
//SYSPRINT    DD  SYSOUT=class
//SYSTERM     DD  SYSOUT=class
//SYSUT1      DD  DISP=(OLD,PASS),DSN=&&NEWQ
//SYSLIN      DD  DISP=OLD,DSN=your.object.library(member)
//
```

Running the compiler without the global optimization phase

The sample JCL in Example Code 5.4 on page 93 illustrates how to run the compiler under OS/390 without the global optimization phase.

Example Code 5.4 Sample JCL for Running the Compiler under OS/390 without the Global Optimization Phase

```
//COMPILE    JOB  job card information
//*-------------------------------------------------------------
//*   EXAMPLE JCL FOR COMPILATION.
//*   REPLACE GENERIC NAMES AS APPROPRIATE.
//*-------------------------------------------------------------
//*   SYNTAX ANALYSIS PHASE
//*-------------------------------------------------------------
//STEP1     EXEC  PGM=LC1370,REGION=1024K,
//    PARM='-R P: -M !X:'
//STEPLIB     DD  DISP=SHR,DSN=compiler.library
//            DD  DISP=SHR,DSN=runtime.library
//SYSLIB      DD  DISP=SHR,DSN=standard.macro.library
//H           DD  DISP=SHR,DSN=your.macro.library
//SYSPRINT    DD  SYSOUT=class
//SYSTERM     DD  SYSOUT=class
//SYSUT1      DD  UNIT=SYSDA,SPACE=(CYL,(1,1)),
//    DISP=(NEW,PASS),
//    DSN=&&QUADS
//SYSIN       DD  DISP=SHR,DSN=your.source.library(member)
//*
```

```
/*
//*-------------------------------------------------------------
//*   CODE GENERATION PHASE
//*-------------------------------------------------------------
//STEP2     EXEC  PGM=LC2370,REGION=1024K,COND=(4,LT)
//STEPLIB     DD  DISP=SHR,DSN=compiler.library
//            DD  DISP=SHR,DSN=runtime.library
//SYSPRINT    DD  SYSOUT=class
//SYSTERM     DD  SYSOUT=class
//SYSUT1      DD  DISP=(OLD,PASS),DSN=&&QUADS
//SYSLIN      DD  DISP=OLD,DSN=your.object.library(member)
//
```

Running the compiler using the debug option

The sample JCL in Example Code 5.5 on page 94 illustrates how to run the compiler under OS/390 using the **debug** option. For more information see the SAS/C Debugger User's Guide and Reference.

Example Code 5.5 Sample JCL for Running the Compiler under OS/390 with debug option

```
//COMPILE    JOB  job card information
//*-------------------------------------------------------------
//*   EXAMPLE JCL FOR COMPILATION.
//*   REPLACE GENERIC NAMES AS APPROPRIATE.
//*-------------------------------------------------------------
//*   SYNTAX ANALYSIS PHASE
//*-------------------------------------------------------------
//STEP1     EXEC  PGM=LC1370,REGION=1024K,
//    PARM='-D -R P: -M !X:'
//STEPLIB     DD  DISP=SHR,DSN=compiler.library
//            DD  DISP=SHR,DSN=runtime.library
//SYSLIB      DD  DISP=SHR,DSN=standard.macro.library
//H           DD  DISP=SHR,DSN=your.macro.library
//SYSPRINT    DD  SYSOUT=class
//SYSTERM     DD  SYSOUT=class
//SYSUT1      DD  UNIT=SYSDA,SPACE=(CYL,(1,1)),
//    DISP=(NEW,PASS),
//    DSN=&&QUADS
//SYSUT2      DD  UNIT=SYSDA,SPACE=(CYL,(1,1)),
//    DISP=(NEW,PASS),
//    DSN=&&SDIF
//SYSIN       DD  DISP=SHR,DSN=your.source.library(member)
//*
/*
//*-------------------------------------------------------------
//*   CODE GENERATION PHASE
//*-------------------------------------------------------------
//STEP2     EXEC  PGM=LC2370,REGION=1024K,COND=(4,LT)
//STEPLIB     DD  DISP=SHR,DSN=compiler.library
//            DD  DISP=SHR,DSN=runtime.library
//SYSPRINT    DD  SYSOUT=class
//SYSTERM     DD  SYSOUT=class
//SYSUT1      DD  DISP=(OLD,PASS),DSN=&&QUADS
```

```
//SYSUT2      DD  DISP=(OLD,PASS),DSN=&&SDIF
//SYSDBLIB    DD  DISP=OLD,DSN=your.debugger.library
//SYSLIN      DD  DISP=OLD,DSN=your.object.library(member)
//
```

Compiler Return Codes

The compiler detects syntax and semantic errors during compilation and generates a return code for error conditions and warnings. These codes are summarized in Table 5.4 on page 95.

Table 5.4 Compiler Return Codes

Code	Meaning
0	No errors or warnings found; object code is generated.
4	Warning: object code is generated and will probably execute correctly.
8	Serious error: object code is generated but may not execute correctly.
12	Serious error: no object code is generated, and pass two of the compiler is not executed.
16	Fatal error: compilation immediately terminates.
20	Fatal error: an abend or internal compiler error occurred. Compilation stops and a dump may be produced.

Note: Under USS, the **-mrc** compiler option requests that the compiler return the same return codes as on OS/390 and CMS. If **-mrc** is not specified, the compiler conforms to UNIX conventions and returns 0 if there were no errors and a non-zero code if errors were detected. △

The Object Module Disassembler

The object module disassembler (OMD) is a useful debugging tool that provides a copy of the assembler code generated for a C program. If the object module is created with a line number-offset table (that is, if the default compiler option **lineno** is in effect), then the C source code is merged with the assembler instructions. Refer to "Object Module Disassembler Options" on page 105 information on OMD options.

Using the OMD in TSO

The following sections describe how to use the OMD in TSO.

The OMD370 CLIST

The OMD370 CLIST runs the object module disassembler independently of compilation. Invoke the object module disassembler with the OMD370 CLIST as follows:

```
OMD370 dsname  [options]
```

dsname is the name of the data set containing the object code to be disassembled and *options* are OMD options. For the data set specification, you must follow standard TSO naming conventions; that is, if the data set belongs to some other user, you must specify the full name of the data set and enclose the full name in single quotes. If the object code is in a member of a partitioned data set, you must specify the member name in parentheses following the data set name, in the normal TSO manner. For example, a data set belonging to another user can be specified as follows:

```
'YOURLOG.PROJ4.OBJ'
```

If a member name is included, the data set can be specified as follows:

```
'YOURLOG.PROJ4.OBJ(PART1)'
```

If you do not enclose the data set name in single quotes, the OMD370 CLIST assumes that the final qualifier of the data set name is OBJ. If you do not specify OBJ, it is added automatically by the CLIST. For example, in the command

```
OMD370 PROJ4(PART1)
```

the C object code is assumed to be in member PART1 of the data set named *prefix*.PROJ4.OBJ, where *prefix* is the user's default prefix. You must separate options with one or more commas, blanks, or tabs. The only requirement is that the object data set name must be the first item on the command line following OMD370. You can abbreviate options to the portion shown in uppercase in Table 6.1 on page 102. To negate an option, precede it with **NO**. If you use both the positive and negative form of an option (for example, **trans** and **notrans**), both options are rejected.

Sample OMD370 CLIST command line with options

Here is a sample command line that invokes the OMD:

```
OMD370 PROJ4(PART1) MER(''ADMIN.PROJ4.SOURCE'') NOTRANS
```

The following items discuss the sample command line:

- The command line invokes the CLIST to disassemble member PART1 of data set *userid*.PROJ4.OBJ, where *userid* represents the ID of the TSO user.
- Since **print** is not specified and the source data set is not enclosed by single quotes, the listing is written to the data set *userid*.PROJ4.LIST.
- The **merge** option specifies that the source for the object module should be read from the data set ADMIN.PROJ4.SOURCE and merged with the OMD output. This data set belongs to another user and is therefore set off by three quotation marks before and after the data set name.
- The **notrans** option specifies that special characters in the source should not be translated or overstruck when they are printed.

Running the OMD as an LC370 CLIST option

The OMD can be executed as an option on the LC370 CLIST. To execute the OMD as a part of compilation, use the option **omd**. If the compiler option **lineno** is allowed to default, the OMD produces generated assembler output merged with the C source code. The format is as follows:

```
LC370 dsname  [options] OMD [OMD options]
```

Using the OMD under CMS

The following sections describe how to use the OMD under CMS.

The OMD370 EXEC

The OMD370 EXEC invokes the object module disassembler without invoking the compiler. You can invoke the OMD with either a CMS fileid or a SAS/C **sf:** style file identifier. The OMD370 EXEC does not accept DDnames. Invoke the OMD using a CMS fileid with one of the following forms:

```
OMD370 filename [filetype] [filemode] [(options[)]]
OMD370 filename.filetype[.filemode] [(options[)]]
```

where *filename*, *filetype* and *filemode* identify the TEXT file to be disassembled, and *options* is any of the CMS options for the OMD shown in Table 6.2 on page 105. If you omit the filetype or filemode, the OMD uses TEXT as the default filetype and an asterisk (*) as the default filemode.

To disassemble an object file which resides in the shared file system, you can use OMD370 specifying a SAS/C **sf:** style file identifier, as follows:

```
OMD370 sf:filename [filetype] [dirname] [(options[)]]
```

where *dirname* is the complete dirname or the NAMEDEF that has been logically assigned to it, and *options* is any of the CMS options for the OMD list in Table 6.2 on page 105. If you omit *filetype* or *dirname*, the default filetype is TEXT; the default dirname is a period (.).

The OMD370 EXEC can also be used to disassemble a module stored as a member of a TEXT library (TXTLIB). In this case, specify the member name as the filename, and specify the TXTLIB name using the **lib** option, as in the following example:

```
OMD370 myprog (lib mylib
```

where MYPROG is a member of MYLIB TXTLIB.

Other OMD options allow you to specify the location of the source code for the disassembled module and how the disassembled output file should be written. By default, the OMD writes a file called *filename* ASM on the A-disk.

The **me** option is used to specify the fileid of the source file which should be merged into the OMD output. The fileid may be either a CMS fileid or an **sf:** style file identifier. The default filetype is C.

If the **me** option is not used, OMD370 looks for the source file on an accessed minidisk if the input fileid was a CMS fileid. If the input fileid was a **sf:** style file identifier, it looks for the source file in the same directory of the input file. The source file is assumed to have a filetype of C and the same filename as the input file.

Several OMD370 options allow you to control the location of the OMD output. The **print** option can be used to write the output to the virtual printer, and the **type** option can be used to write the output to the terminal.

The **pr** option allows you to specify the fileid of the OMD's output file. The fileid may be either a CMS fileid or a SAS/C **sf:** style file identifier. The default filetype is ASM. Note that the **type** and **print** options are ignored when **pr** is specified.

If neither **print**, **type**, nor **pr** is specified, OMD370 writes its output file to a file whose filetype is ASM and whose filename is the same as the input file's. If the input file was specified using a CMS fileid, the ASM file is written to the minidisk containing the input file, or to the A-disk if the input minidisk is not writable. If the input file was specified using a **sf:** style file identifier, the ASM file is written to the input file directory if it is writable, or to the top directory if the input file directory is not writable.

Running the OMD as an LC370 EXEC option

The OMD can be executed as an option on the LC370 EXEC. To execute the OMD as part of compilation, use the **omd** option. The option produces generated assembler language output based on the compiler-produced object code. If the **lineno** option has

been used, the C source code is merged with the assembler language statements. When the OMD option to the LC370 EXEC is used, any OMD options also can be used.

Using the OMD under OS/390 Batch

The following sections describe how to use the OMD under OS/390 batch.

The LC370D cataloged procedure

The procedure LC370D runs the OMD independently of compilation. When you use LC370D, you need DD cards for SYSLIN (your object data set), and if you are running with the **merge** option, you need DD cards for SYSIN (your source data set). See Example Code 5.6 on page 98 for typical JCL. See Chapter 6, “Compiler Options,” on page 101 for detailed information about the OMD options.

Example Code 5.6 Sample JCL for Running the LC370D Cataloged Procedure

```
//DISASMBL JOB   job card information
//*-------------------------------------------------------
//*   EXAMPLE JCL FOR DISASSEMBLY
//*   REPLACE GENERIC NAMES AS APPROPRIATE.
//*-------------------------------------------------------
//STEP1   EXEC  LC370D,PARM.D='options '
//D.SYSLIN  DD  DISP=SHR,DSN=your.object.library(member)
//D.SYSIN   DD  DISP=SHR,DSN=your.source.library(member)
//
```

SYSLIN or SYSIN or both may reference USS HFS files.

Example Code 5.7 on page 98 shows the JCL for the procedure LC370D. This JCL is correct as of the publication of this guide. However, it may be subject to change. 1ln

Example Code 5.7 Expanded JCL for LC370D

```
//LC370D PROC
//*************************************************************
//*  PRODUCT:  SAS/C                                          ***
//*  PROCEDURE:  DISASSEMBLY                                  ***
//*  DOCUMENTATION:  SAS/C COMPILER AND LIBRARY USER'S GUIDE   ***
//*  FROM:  SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC     ***
//*************************************************************
//*
//D        EXEC PGM=LC370DM
//STEPLIB  DD DSN=SASC.LOAD,
//            DISP=SHR        COMPILER LIBRARY
//         DD DSN=SASC.LINKLIB,
//            DISP=SHR       RUNTIME LIBRARY
//SYSTERM  DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
```

OMD JCL requirements

This section discusses the data definition statements needed to run the OMD under OS/390 if you are writing your own JCL. As is the case with compiler options, OMD

options must be given in their short forms when OMD370 is invoked directly. You need the data definition (DD) statements shown in Table 5.5 on page 99 to invoke the OMD.

Table 5.5 Data Definition Statements Used by the OMD

DDname	Purpose
SYSIN	C source matching the SYSLIN input
SYSLIN	object module to be disassembled
SYSPRINT	standard output
SYSTERM	error output

SYSIN can be specified as DD DUMMY if the source code is not available or if a merged listing is not required.

Example Code 5.8 on page 99 illustrates how to invoke the OMD.

Example Code 5.8 Sample JCL for Running the OMD

```
//JOBNAME  JOB  job card information
//*
//OMD      EXEC PGM=OMD370,PARM='options '
//STEPLIB     DD  DISP=SHR,DSN=compiler.library
//            DD  DISP=SHR,DSN=runtime.library
//*
//SYSIN    DD  DSN=your.source.library(member) ,DISP=SHR
//SYSLIN   DD  DSN=your.object.library(member) ,DISP=SHR
//SYSPRINT DD  SYSOUT=class
//SYSTERM  DD  SYSOUT=class
```

Note that you can easily run the OMD when a module is being compiled using the LC370C cataloged procedure.

Running the OMD with LC370C

The OMD can be executed as an option in LC370C. To execute the OMD as a part of compilation, use the option **omd**. If the compiler option **lineno** is allowed to default, the OMD produces generated assembler output merged with the C source code. You can also use any OMD options, for example:

```
// EXEC  LC370C,PARM.C='TRANS,PRINT,OMD,VERBOSE'
```

In this example, **trans** and **print** are compiler options, **omd** invokes the OMD after the compiler has completed, and **verbose** is an OMD option. See Chapter 6, "Compiler Options," on page 101 for more information on compiler options and OMD options.

OMD options (short forms)

The OMD takes seven short-form options, as summarized in Table 5.6 on page 100. The options can be in upper- or lowercase. As with the short forms of the compiler options, precede positive forms of the options with a hyphen (-) as the initial character; precede negative forms with an exclamation point (!) or a not sign (¬). For simplicity, the table indicates only the positive form of the option. Refer to Chapter 6, "Compiler Options," on page 101 for complete descriptions of these options.

Table 5.6 OMD Option Equivalents

Long Form	Short Form	Long Form	Short Form
c	**-c**	**japan**	**-j**
merge	**-s**	**overstrike**	**-to**
trans	**-t**	**upper**	**-u**
verbose	**-v**		

The following is an example of an EXEC statement that invokes the OMD with the options **notrans** and **verbose**:

```
// EXEC PGM=OMD370,PARM='!T -V'
```

CHAPTER

6

Compiler Options

Introduction

The SAS/C Compiler accepts a number of options enabling you to alter the way code is generated, the appearance of listing files, and other aspects of compilation. This chapter explains what options are available and how to specify them in TSO, under CMS, and under OS/390.

Since the object module disassembler (OMD) is often executed as part of compilation, the options accepted by the OMD are also discussed in this chapter.

Option Summary

Table 6.1 on page 102 summarizes all compiler options. The option name is in the first column. Capital letters indicate the abbreviation for the option. The second column lists the default for each option. For the default of some options, you are referred to the description of the option later in the chapter. The third column indicates how the option is specified from the UNIX System Services (USS) shell. The fourth column indicates whether the option can be negated. An exclamation point (!) means that the option can be negated. A plus sign (+) means that the option cannot be negated. (To negate an option under OS/390 or CMS, precede the option name with NO. To negate an option under the shell, insert **no** after the **-K** beginning the option name. For instance, to negate the IMULTI option under OS/390 or CMS, use NOIMULTI. Under USS, use **-Knoimulti**.) The next three columns list the environment(s) for which an option is implemented. The Affects Process column names the process that the option affects as follows:

- C compilation
- O object module disassembler (OMD)
- L listing

- X cross-referencing
- M message generation
- G global optimization

An asterisk (*) under the Sys column means that the form or meaning of the option may differ depending on the environment in which the compiler is running. Detailed information about the options follows the table.

Note: Under OS/390 batch, the USS shell, and CMS, if you specify contradictory options, the option specified last is used. In TSO, the options are concatenated and treated as a single invalid option. △

Table 6.1 Compiler Options

Option Name	Default	USS	Negation	OS/390 Batch	TSO	CMS	Affects Process	Sys
ALias	**NOALias**	**-Kalias**	!	*	*	*	G	
ARlib	see description		+		*		C	*
ARMode	**NOARMode**	**-Karmode**	!	*	*	*	C	
ASciiout	**NOASciiout**	**-Kasciiout**	!	*	*	*	C	
AT	**NOAT**	**-Kat**	!	*	*	*	C	
AUtoinst	**NOAUtoinst**	**-Kautoinst**	!	*	*	*	C	
BItfield	**NOBItfield**	**-Kbitfield=*n***	!	*	*	*	C	*
BYtealign	**NOBYtealign**	**-Kbytealign**	!	*	*	*	C	
C	**C**	**-c**	!	*	*	*	O	
			+				C	
COMNest	**NOCOMNest**	**-Kcomnest**	!	*	*	*	C	
COMPlexity	**COMPlexity(0)**	**-Kcomplexity=*u***	+	*	*	*	G	*
CXX	**NOCXX**	**-cxx**	!	*	*	*	C	
DBHook	**NODBHook**	**-Kdbhook**	!	*	*	*	C	
DBGLib	see description		+		*		C	*
DBGMacro	**NODBGMacro**	**-Kdbgmacro**	!	*	*	*	C	
DBGObj	**NODBGObj**	**-Kdbgobj**	!	*	*	*	C	
DEBug	**NODEBug**	**-Kdebug[=*filename*]**	!	*	*	*	C	
DEFine	see description	**-D[*sym*=*val*]**	!	*	*	*	C	*
DEPth	**DEPth(3)**	**-Kdepth=*n***	!	*	*	*	C	*
DIgraph	see description	**-Kdigraph[n]**	!	*	*	*	C	
DIsk	see description		!			*	C	*
DOllars	**NODOllars**	**-Kdollars**	!	*	*	*	C	
ENForce	see description	**-w~*n***	+	*	*	*	M	*
ENXref	**NOENXref**	**-Kenxref**	!	*	*	*	C	
EXClude	**EXClude**	**-Kexclude**	!	*	*	*	L	

Option Name	Default	USS	Negation	OS/ 390 Batch	TSO	CMS	Affects Process	Sys
EXTname	see description	**-Kextname**	!	*	*	*	C	*
FIles	see description		+	*			C, O	*
FReg	**FReg(2)**	**-Kfreg=*n***	!	*	*	*	G	*
GLobal	**GLobal**		!			*	C	*
GReg	**GReg(6)**	**-Kgreg=*n***	+	*	*	*	G	*
HList	**NOHList**	**-Khlist**	!	*	*	*	L	
HMulti	**HMulti**	**-Kmulti**	!	*	*	*	C	
HXref	**NOHXref**	**-Khxref**	!	*	*	*	X	
IGline	**NOIGline**	**-Kigline**	!	*	*	*	C	
IList	**NOIList**	**-Kilist**	!	*	*	*	L	
IMulti	**IMulti**	**-Kimulti**	!	*	*	*	C	
INDep	**NOINDep**	**-Kindep**	!	*	*	*	C	
INLine	**INLine**	**-Kinline**	!	*	*	*	G	
INLOcal	**NOINLOcal**	**-Kinlocal**	!	*	*	*	G	
IPath	see description	**-I*pathname***	+	*	*	*	C	*
IXref	**NOIXref**	**-Kixref**	!	*	*	*	X	
Japan	**NOJapan**	**-Kjapan**	!	*	*	*	C, L, X, M, O	
-Knobinder	**-Kbinder**	**-Knobinder**						
LIB	**NOLIB**		!		*	*	C	*
LINeno	**LINeno**	**-Klineno**	!	*	*	*	C	
LOop	**LOop**	**-Kloop**	!	*	*	*	G	
MAClist, **MList**	**NOMAClist**, **NOMList**	**-Kmaclist**	!	*	*	*	L	
ME	see description		+			*	O	*
MEMber	see description		+		*		C	*
MENtion	see description	**-w+*n***	+	*	*	*	M	*
MERge	**MERge**	**-mrc**	!	*	*	*	O	*
			+				C	
OBject	see description	**-o *filename***	!		*	*	C	*
OMD	**NOOMD**	**-Komd[=*filename*]**	!	*	*	*	O	
OPTIMize	**NOOPTIMize**	**-Koptimize**	!	*	*	*	G	
OPTIOns	**OPTIOns**	**-Koptions**	!	*	*	*	L	
OVERStrike	**NOOVERStrike**	**-Koverstrike**	!	*	*	*	L, X, O	

Option Name	Default	USS	Negation	OS/390 Batch	TSO	CMS	Affects Process	Sys
PAgesize	**PAgesize(60)**	**-Kpagesize=*nn***	+	*	*	*	L, X	*
PFLocal	**NOPFLocal**	**-Kpflocal**	!	*	*	*	C	
POsix	see description	**-Kposix**	!	*	*		C	*
PPIx	**NOPPIx**	**-Kppix**	!	*	*	*	C	
PPOnly	**NOPPOnly**	**-P**	!	*	*	*	C	*
PR	see description		+			*	O	*
PRInt	see description	**-Klisting[=*filename*]**	!	*	*	*	L, X, O	*
RDEpth	**REDept(1)**	**-Krdepth=*n***	+	*	*	*	G	*
REDef	**NOREDef**	**-Kredef**	!	*	*	*	C	
REFDef	**NOREFDef**	**-Krefdef**	!	*	*	*	C	
RENT	**NORENT**	**-Krent**	!	*	*	*	C	
RENTExt	**NORENTExt**	**-Krentext**	!	*	*	*	C	
REQproto	**NOREQproto**	**-cf**	!	*	*	*	C	
SMPxivec	**NOSMPxivex**	**-Ksmpxivec**	!	*	*	*	C	
SName	see description	**-Ksname=*sname***	+	*	*	*	C	*
SOurce	**SOurce**	**-Ksource**	!	*	*	*	L	
STMap	NOSTMap	-Kstmap	!	*	*	*	X	
STRICt	**NOSTRICt**	**-Kstrict**	!	*	*	*	M	
STRINgdup	**STRINgdup**	**-Kstringdup**	!	*	*	*	C	
SUppress	see description	**-w*n***	+	*	*	*	M	
		-temp=*directory*	+				C	
TErm	see description		!	*	*	*	M	*
TRAns	**TRAns**	**-Ktrans**	!	*	*	*	L, X, O	
TRIgraphs	**NOTRIgraphs**	**-Ktrigraphs**	!	*	*	*	C	
TYpe	see description		+			*	C	*
UNdef	**NOUNdef**	**-Kundef**	!	*	*	*	C	*
UPper	**NOUPper**	**-Kupper**	!	*	*	*	L, X, O	
USearch	see description	**-Kusearch**	!	*	*	*	C	*
-v			+				C	
VErbose	**NOVErbose**		!	*	*	*	O	
VString	**NOVString**	**-Kvstring**	!	*	*	*	C	
Warn	**Warn**	**-Kwarn**	!	*	*	*	M	
Xref	**Xref**	**-Kxref**	!	*	*	*	X	

Option Name	Default	USS	Negation	OS/ 390 Batch	TSO	CMS	Affects Process	Sys
ZAPMin	**ZAPMin(24)**	**-Kzapmin=*n***	+	*	*	*	C	*
ZAPSpace	**ZAPSpace(1)**	**-Kzapspace=*n***	+	*	*	*	C	*

Object Module Disassembler Options

Table 6.2 on page 105 shows OMD options from Table 6.1 on page 102. Not all of these options are valid for every operating environment.

Table 6.2 OMD Options

Compiler and OMD Options	OMD-only Options
disk	**c**
files	**me**
japan	**merge**
overstrike	**pr**
print	**verbose**
trans	
type	
upper	

If you use the **OMD** option to run the object module disassembler at compile time, the options in the first column in Table 6.2 on page 105 affect the OMD (and other processes). For example, used on the command line, **overstrike** affects any OMD listing you request, as well as any other listings you ask for.

The options shown in the second column apply to the object module disassembler only.

If you run the OMD independent of compilation, you can use all of the options with an O in the Affects Process column in Table 6.1 on page 102 and summarized in Table 6.2 on page 105, subject to system dependencies. (For example, the **files** option is valid for OS/390 batch only.) For details, refer to each option description on the following pages and to the description of command lines to invoke the compiler and OMD for your operating system.

Option Descriptions

alias (**-Kalias** under USS)
specifies that the compiler should assume worst-case aliasing.
See “The optimize Option” on page 63 for details about this option. This option can be used only with the **optimize** option.

arlib
under TSO, specifies an AR370 archive that is used to store the compiler's output. The form of this option is as follows:

```
arlib(archive)
```

The *archive* parameter specifies the data set name of the AR370 archive. 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 single quotes. For example:

```
'''userid.archive-name.AR'''
```

If the archive name is not quoted and does not have a final qualifier of AR, a final qualifier of AR is appended. **arlib** may be used in conjunction with the **member** option, which specifies the AR370 library member name. If you do not use the **member** option, the LC370 CLIST will use the member name of your source file if possible or prompt you to enter the member name.

armode
specifies that code that uses the ESA access registers may be generated. This option is required to compile code that uses far pointers. See "Optimization and Far Pointers" on page 79 for more information on far pointers and access register mode.

asciiout (**-Kasciiout** under USS)
requests ASCII translation of character and string literals. The default is **noasciiout**, and the minimum abbreviation is **as**. When the **asciiout** option is used, the compiler generates string literals and character literals using the ASCII character set instead of the default EBCDIC character set. String literals are translated from IBM Code Page 1047 to ISO 8559-1, the Latin-1 character set.

Use the **asciiout** option with extreme care. The run-time library expects all string and character literals to be in EBCDIC format. Therefore, when using the **asciiout** option, you should avoid calls to library functions that pass character or string literals unless great care is taken. For example, calls to **printf**, **scanf,** and so on that use literal format strings may not produce the intended results because the functions cannot interpret an ASCII format string.

```
sprintf(buffer, "Number of %s was %d.\n", things, n);
                          /* This won't work if ASCII is used! */
```

at (**-Kat** under USS)
allows the use of the call-by-reference operator **@**.

autoinst (**-Kautoinst** under UNIX System Services)
The **autoinst** option controls automatic implicit instantiation of template functions and static data members of template classes. The compiler organizes the output object module so that COOL can arrange for only one copy of each template item to be included in the final program. In order to correctly perform the instantiation, the **autoinst** option must be enabled on a compilation unit that contains both a use of the item and its corresponding template identifier. (See the SAS/C C++ Development System User's Guide for information about templates and automatic instantiation.)

Note: COOL must be used if this option is specified. △

bitfield *n* (**-Kbitfield=***n* under USS)
allows bitfields that are not **int** and specifies an allocation unit. This option requires that you specify the allocation unit *n* to be used for plain **int** bitfields. The values can be either 1, 2, or 4, which specifies that the allocation unit be a

char, **short**, or **long**, respectively. See "Noninteger bitfields" on page 34 for more details.

In TSO and under OS/390 batch, the **bitfield** option is specified as follows:

```
bitfield(value)
```

For example, the following indicates that bitfields that are not **int** are accepted and that the allocation unit for **int** bitfields is a **short**:

```
bi(2)
```

Under CMS, the **bitfield** option is specified as follows:

```
bitfield value
```

For example, the following indicates that bitfields that are not **int** are accepted and that the allocation unit for **int** bitfields is a **long**:

```
bitfield 4
```

bytealign (**-Kbytealign** under USS)
aligns all data on byte boundaries. Most data items, including all those in structures, are generated with only character alignment. Because formal parameters are aligned according to normal IBM 370 conventions, even with the **bytealign** option, you can call functions compiled with byte alignment from functions that are not compiled with byte alignment and vice versa.

If functions compiled with and without byte alignment are to share the same structures, you must ensure that such structures have exactly the same layout. The layout is not exactly the same if any structure element does not fall on its usual boundary, for example, an **int** member's offset from the start of the structure is not divisible by 4. You can force such alignment by adding unreferenced elements of appropriate length between elements, as necessary. If a shared structure does contain elements with unusual alignment, you must compile all functions that reference the structure with the byte alignment option.

c
identifies the object code to be disassembled as code generated by the compiler. The negative (**noc**) means that the object code to be processed by the OMD was not generated by the compiler. **c** is an OMD-only option. See also **omd** and **object**.

-c (USS only)
specifies that only the compiler is to be run. After completion of compilation, the prelink and link steps are bypassed. Normally, these steps follow compilation under USS.

comnest (**-Kcomnest** under USS)
allows nested comments.

complexity (**-Kcomplexity=***n* under USS)
specifies the maximum complexity the function can have and remain eligible for default inlining for functions that have not been defined using the **__inline** keyword. Used with **optimize** only.

Specify the **complexity** option as follows, where *n* is a value between 0 and 20 inclusive:

- under OS/390 batch: **complexity(*n*)**
- in TSO: **complexity(*n*)**
- under CMS: **complexity *n***

See "The optimize Option" on page 63 for more details.

cxx (**-cxx** under USS)
specifies that the source code being compiled is generated by the SAS/C++ translator. For more information on the **cxx** option, refer to the SAS/C C++ Development System User's Guide.

dbhook (**-Kdbhook** under USS)
generates hooks in the object code that is generated by the compiler. When you compile a module with the **debug** option, the **dbhook** option is implied. **dbhook** can be used with the **optimize** option to enable limited debugging of optimized object code. The default is **nodbhook**. See "The optimize Option" on page 63 for more details.

dbglib
specifies a debugger file qualifier that provides for customization of the destination of the debugger file.

For each platform, **dbglib** specifies something different:

ON MVS:
A SAS/C file specification that denotes a PDS. The filename is constructed using whatever is supplied, followed by (***sname***).

On CMS:
If the option specified starts with a '**/**', then it is assumed that this is either a '**//sf:**' file specification or an SFS path. In this case, the specification is prepended to the filename. For example,

```
dbglib(//sf:ted/)
```

will generate the name

```
//sf:/ted/sname.DB
```

If the option specified does not start with a '**/**' then it is considered to be a filemode, and will be appended to the filename. For example,

```
dbglib(d2)
```

will generate the name

```
sname.db.d2
```

On USS:
The option specified is a path name to be prepended to the filename. For example,

```
dbglib(/u/sasc/dbg/)
```

will generate a filename of

```
/u/sasc/dbg/sname.dbg370
```

The option has different defaults on the various platforms:

ON MVS:
dbglib(ddn:sysdblib)

On CMS:
dbglib(A)

On USS:
dbglib()

For the various platforms, the default filename has different forms:

On MVS:
ddn:sysdblib(*sname*)

On CMS:

***sname*.DB.A**

On USS:

hfs:*sname*.dbg370

Note: On USS and UNIX platforms, the sname is capitalized and remains so for debugger filename generation. △

The short form of this option is **-db**.

dbgmacro (**-Kdbgmacro** under USS)
specifies that definitions of C macro names should be saved in the debugger file. Note that this substantially increases the size of the file.

dbgobj (**-Kdbgobj** under OpenEditon)
The **dbgobj** option causes the compiler to place the debugging information in the output object file, instead of a separate debugger file. If the debugging information is not placed in the object file, you cannot debug the automatically instantiated objects.

If automatic instantiation is specified with the **autoinst** option, **dbgobj** is enabled automatically.

By default, the **dbgobj** option is off. The short form for the option is **-xc**. See the SAS/C C++ Development System User's Guide for information about templates and automatic instantiation.

Note: COOL must be used if this option is specified. △

debug (**-Kdebug[=***filename***]** or **-g** under USS)
allows the use of the debugger to trace the execution of statements at runtime. For programs not compiled with **debug**, only calls can be traced. Note that if you use **debug**, **lineno** is implied. Also note that the **debug** option causes the compiler to suppress all optimizations as well as store and fetch variables to or from memory more often.

Under the USS shell, you can supply the name of the debugger symbol table file by specifying **-Kdebug=***filename*. If **-Kdebug** without a filename or **-g** is specified, the debugger file is stored in an HFS file with a **.dbg370** extension. The name of the default debugging file is derived from the source filename in the same way as the object file, except for the **.dbg370** extension. See the **object** option for a description of this process. Note that you should not specify an explicit filename if you compile more than one source file at a time, since each individual compilation will overwrite the debugging file.

define (**-D[***sym*=*val***]** under USS)
defines a symbol with an optional value. The following are further instructions for using **define** in different environments.

- Under OS/390 batch, the **define** option is specified as follows:

```
define(symbol)
```

or

```
define(symbol=value)
```

The following is an example of defining a symbol:

```
def(USERDATA)
```

Under OS/390 batch, you can use the **define** option more than once to define more than one symbol.

- In TSO, the specification is the following:

```
define(symbol)
```

or

```
define(symbol=value)
```

The following is an example of defining a symbol:

```
define(TSO=1)
```

In TSO, the **define** option can be used only once. If you specify this option more than once, only the last specification is used. Note that the TSO CLIST language automatically converts both the symbol and the value to uppercase.

- Under CMS, the specification is the following:

```
define symbol
```

or

```
define symbol=value
```

The following are examples of defining a symbol:

```
define USERDATA
```

and

```
define CMS=1
```

Under CMS, you can use the **define** option more than once to define more than one symbol.

- Under the USS shell, the specification is the following:

```
-Dsymbol
```

or

```
-Dsymbol=value
```

The following are examples of defining a symbol:

```
-DUSERDATA
```

and

```
-DUNIX=1
```

Under the shell, you can use the **-D** option more than once to define more than one symbol.

depth (**-Kdepth=***n* under USS)
specifies the maximum depth of function calls to be inlined. **depth** is used with **optimize** only. See "The optimize Option" on page 63 for more information.

Specify **depth** as follows, where *n* is between 0 and 6 inclusive (the default is 3):

- under OS/390 batch: **depth(*n*)**
- in TSO: **depth(*n*)**
- under CMS: **depth *n***

digraph (**-Kdigraph** under UNIX System Services)
Digraph options enable the translation of the International Standard Organization (ISO) digraphs and/or the SAS/C digraph extensions.

A digraph is a two character representation for a character that may not be available in all environments. The different options allow you to enable subsets of the full diagraph support offered collectively by ISO and SAS/C. Table 6.3 on page 111 gives a brief description of the new digraph compiler options.

Table 6.3 Digraph Descriptions

Digraph No.	Description
0	Turn off all digraph support
1	Turn on New ISO digraph support
2	Turn on SAS/C Bracket digraph support - '(|' or '|)'
3	Turn on all SAS/C digraphs.

Table 6.4 on page 111 provides the default values and an example of how to negate the options in each of the different environments.

Table 6.4 Digraph Default and Negated Forms

Environment	Default Options	Negated Options
IBM 370 (Long Form)	**DI(1)**, **DI(3)**	**NODI(1)**, **NODI(3)**
IBM 370 and Cross (Short Form)	**-cgd1**, **-cgd3**	**!cgd1**, **!cgd3**
Cross Compiler and IBM 370 UNIX System Services	**-Kdigraph1**, **-Kdigraph3**	**!Kdigraph1**, **!Kdigraph3**

Table 6.5 on page 111 lists several of the ISO digraph sequences from the C++ ANSI draft. Basically, the alternative sequence of characters is an alternative spelling for the primary sequence. Similar to SAS/C digraphs, substitute sequences are not replaced in either string constants or character constants. SAS/C Release 7.00 supports the left column of pairs of primary and alternative sequences.

Table 6.5 ISO digraph Alternative Tokens

Rel 6.50 Tokens	
Primary	**Alternate**
{	<%
}	%>
[	<:
]	:>
#	%:
##	%:%:

Note: See "Special Character Support" on page 20 for more information on digraphs. △

disk
writes the listing file on the A-disk. **disk** is a valid option under CMS only. When using the **disk** option, remember the following:

- If you request **disk** but do not specify **term** or **noterm**, the default is **term**; the listing file goes to the disk, and error messages are also sent to the terminal.
- If you do not specify **print**, **noprint**, or **type**, the default is **disk**. (Note that these options are also mutually exclusive.)

See also **print**, **term**, and **type**, and refer to Table 6.6 on page 126 and Table 6.8 on page 127. If more than one of the **type**, **print**, or **disk** options is specified, the last one entered is in effect.

dollars (**-Kdollars** under USS)
allows the use of the dollar sign ($) character in identifiers, except as the first character.

enforce (**-w~***n* under USS)
treats one or more warning conditions as error conditions. Each warning condition is identified by its associated message number. Only warnings in the ranges 0-199 and 300-499 are affected. Conditions whose numbers have been specified are treated as errors, and the compiler return code is set to 12 instead of 4.

The following are further instructions for using **enforce** in different environments:

- In TSO, specify the **enforce** option as follows:

  ```
  enforce(n)
  ```

  where *n* is the number of the message associated with the warning condition. If more than one warning condition is to be enforced, specify each number in a comma-delimited list, enclosed by quotes, as follows:

  ```
  enforce('n1,n2,...')
  ```

  Any number of warning conditions can be specified. If both **suppress** and **enforce** specify the same warning message number, the warning is enforced.
- Under CMS, use the following:

  ```
  enforce n
  ```

  or

  ```
  enforce n1 n2 ...
  ```
- Under OS/390 batch, use the following:

  ```
  enforce(n)
  ```

  or

  ```
  enforce(n1,n2,...)
  ```
- Under the USS shell, use the following:

  ```
  -w~n
  ```

  or

  ```
  -w~n1 -w~n2 ...
  ```

enxref (**-Kenxref** under USS)
causes the compiler to generate two extended names cross-references. The first cross-reference is in alphabetical order by C identifier (the name in the source file). The second cross-reference is in alphabetical order by link id, which is the

@@*xxxxxx* form assigned by the compiler. The default is **noenxref**. See also the **extname** option description.

exclude (**-Kexclude** under USS)
omits listing lines from the formatted source that are excluded by **#if**, **#ifdef**, and so on. For example, in the following sequence

```
#ifdef MAX_LINE
   printf("Line overflow\n");
#endif
```

the **exclude** option omits the **printf** statement from the formatted source listing if **MAX_LINE** is not currently defined with the **#define** command.

extname (**-Kextname** under USS)
enables the use of extended names. The default is **noextname** for TSO, CMS, and OS/390 batch and **-Kextname** for USS.

The 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 **extern** variable or that identifies an **extern** or **static** function and fits either of the following criteria:

- is greater than eight characters long
- is eight characters or fewer in length but contains uppercase alphabetic characters and is not the name of an **__asm** or high-level language (for example, **__pascal**) function.

Note: If you specify the **extname** option, be sure to include the appropriate header files for any library functions that you use. Some library functions, such as **localtime** and **setlocale**, are more than eight characters in length 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. For example, **<time.h>** contains the following statement: △

```
#pragma map(localtime, "#LOCALTM")
```

This statement converts **localtime** to a shorter name that the compiler does not treat as extended. If you do not include the appropriate header file for a library function with a long name, the compiler treats it as an extended name and generates a reference to the extended name that cannot be resolved from the standard library. For more information on **#pragma map**, refer to "The #pragma map statement" on page 37.

files
replaces SYS in compiler DDnames with the prefix *xxx*. This option is valid for OS/390 batch only.

The only DDname in which SYS cannot be replaced is SYSTERM. The *xxx* prefix can contain from one to three characters. For example,

```
files(job)
```

substitutes the DDnames in column 1 with those in column 2, as follows:

column 1	column 2
SYSIN	JOBIN
SYSPRINT	JOBPRINT
SYSLIN	JOBLIN

column 1	column 2
SYSLIB	JOBLIB
SYSUT1	JOBUT1
SYSUT2	JOBUT2
SYSUT3	JOBUT3
SYSDBLIB	JOBDBLIB
SYSPROTO	JOBPROTO
SYSPPOUT	JOBPPOUT

freg (**-Kfreg=***n* under USS)
specifies the maximum number of floating-point registers that the compiler can assign to register variables in a function. **freg** is used with **optimize** only. The format is

```
freg n
```

where *n* is 0 to 2 inclusive (the default is 2). Under OS/390 and TSO, **freg** is coded as **freg(***n***)**; under CMS it is coded as **freg** *n*. See "The optimize Option" on page 63 for additional details.

global
invokes default values for MACLIBs set with the CMS GLOBALV command. This option is valid under CMS only.

If you specify **noglobal**, automatic reference to the GLOBALV variable MACLIBs is suppressed. However, the GLOBALV variable OPTIONS is still used.

greg (**-Kgreg=***n* under USS)
specifies the maximum number of registers that the compiler can assign to register variables in a function. **greg** is used with **optimize** only. The format is **greg n** where *n* is 0 to 6 inclusive (the default is 6). Under OS/390 and TSO, **greg** is specified as **GREG(n)**. Under CMS, it is specified as **greg** *n*. See "The optimize Option" on page 63 for more details.

hlist (**-Khlist** under USS)
includes standard header files in the formatted source listing. These are files that are included using the following syntax:

```
#include <name.h>
```

or

```
#include <name>
```

hmulti (**-Khmulti** under USS)
specifies that system header files should be included each time they are referenced by a **#include** statement, even if the same file has previously been included. If **nohmulti** is specified, the compiler only includes one copy of the header file code even if the header file is specified by more than one **#include <***filename***>** statement. **hmulti** is the default.

hxref (**-Khxref** under USS)
prints references in standard header files in the cross-reference listing. See **hlist** for a description of header files.

igline (**-Kigline** under USS)
causes the compiler to ignore any **#line** statements in the input file. The default is **noigline**.

ilist (**-Kilist** under USS)
includes user header files referenced by the **#include** statement in the formatted source listing. The **#include** filename appears in the right margin of each line taken from the **#include** file. See also **hlist**.

imulti (**-Kimulti** under USS)
specifies that user header files should be included each time they are referenced by a **#include** statement, even if the same file has previously been included. If **noimulti** is specified, the compiler only includes one copy of the header file code even if the header file is specified by more than one **#include "*filename*"** statement. **imulti** is the default.

indep (**-Kindep** under USS)
generates code that can be called before the C framework is initialized or code that can be used for interlanguage communication. See Chapter 14, "Systems Programming with the SAS/C Compiler," on page 273 and Appendix 6, "Using the indep Option for Interlanguage Communication," on page 393 for a detailed description of the use of this option.

inline (**-Kinline** under USS)
inlines small functions identified by **complexity** and those with **__inline** keyword. **inline** is used with **optimize** only. See "The optimize Option" on page 63 for more details.

inlocal (**-Kinlocal** under USS)
inlines single-call static functions. **inlocal** is used with **optimize** only. See "The optimize Option" on page 63 for more information.

ipath (**-I***pathname* under USS)
specifies a location that is to be searched for header files. The pathname may specify an HFS directory, an OS/390 PDS, or a CMS shared-file system directory. If you are running under the USS shell, the pathname is assumed to be an USS HFS directory unless you precede the name with two slashes and possibly a SAS/C style prefix. If you are running under OS/390 or CMS, a style prefix is required as part of the pathname unless you are referring to an OS/390 DDname. See "Include-File Processing" on page 11 for more information on the **ipath** option. Note that under TSO you can only specify the **ipath** option once, and that the pathname is automatically converted to lowercase.

ixref (**-Kixref** under USS)
lists references in user **#include** files.

japan (**-Kjapan** under USS)
translates keywords and identifiers that are in uppercase to lowercase before they are processed by the compiler. It prints messages in uppercase. This option is intended to be used with terminals or printers that support only uppercase (Roman) characters.

-Knobinder (UNIX System Services only)
Under USS, by default, the Binder is automatically invoked if the output from COOL was successful. Use the **-Knobinder** option to prevent the Binder from executing. If COOL runs successfully, an output file from COOL will be saved. **-Kbinder** is the default.

lib
identifies a header file library.
In TSO, the **lib** option is specified as follows:

```
lib(dsname)
```

where *dsname* indicates the name of a library that contains header files, that is, one containing members that are to be included using the **#include <*member*.h>** (or **<*member*>**) form of the **#include** statement. If the library belongs to another user, the fully qualified name of the data set must be used, and the name must be preceded and followed by three single quotes (because of CLIST language requirements). No final qualifier is assumed for a **lib** data set. **nolib** is the default. **nolib** indicates that no header file libraries are required other than the standard library provided with the compiler.

lineno (**-Klineno** under USS)
enables identification of source lines in run-time messages. When **lineno** is specified, module size is increased because of the generation of line number and offset tables. **lineno** is the default.

loop (**-Kloop** or **-Ol** under USS)
specifies that the compiler should perform loop optimizations. Use this option for multitrip loops. See "The optimize Option" on page 63 for details on this option. This option can be used only in conjunction with the **optimize** option.

maclist or **mlist** (**-Kmaclist** under USS)
prints macro expansions. Source code lines containing macros are printed before macro expansion.

me
under CMS, specifies the fileid of the source file used by the object module disassembler. The form of the **me** option is as follows:

```
me(source-fileid)
```

where *source-fileid* is a CMS fileid or SAS/C **sf:** style filename. The default filetype is C. See "Using the OMD under CMS" on page 96.

member
under TSO, the **member** option is used with the **arlib** option to specify the output archive member name. The form of the option is as follows:

```
ar370(archive) member(member)
```

where *archive* specifies the AR370 archive and *member* specifies the member to store the output in.

mention (**-w+***n* under USS)
specifies that the warnings whose numbers are specified as *n1*, *n2*, and so on, are not to be suppressed. Only warnings in the ranges 0-199 and 300-499 are affected. See also **suppress**.

- In TSO, specify the **mention** option as follows:

  ```
  mention(n)
  ```

  where *n* is the number of the message associated with the warning condition. If more than one warning condition is to be mentioned, specify the numbers in a comma-delimited list, enclosed by quotes, as follows:

  ```
  mention('n1,n2,...')
  ```

- Under CMS, use the following:

  ```
  mention n
  ```

  or

```
mention n1 n2 ...
```

- Under OS/390 batch, use the following:

```
mention(n)
```

or

```
mention(n1,n2 ...)
```

- Under the USS shell, use the following:

```
-w+n
```

or

```
-w+n1 -w+n2 ...
```

Any number of warning conditions can be specified, regardless of the environment.

merge
merges a copy of the source code into the OMD listing.

Under OS/390 batch, when used as an option with the appropriate PROC, **merge** merges the source code into the OMD listing of the object code.

In TSO, this option is specified as follows:

```
merge(dsname)
```

where *dsname* names the data set from which the OMD is to read the source code for the module to be disassembled. The data set name can be a sequential data set or a partitioned data set member. If the data set belongs to another user, the fully qualified name of the data set must be specified, and the name must be preceded and followed by three single quotes, as in the following example:

```
OMD370 PROJ4(PART1) MER('''YOURLOG.PROJ4.SOURCE(PART1)''')
```

The extra quotes are required for the CLIST language. If the data set name is not specified within three single quotes, it is assumed to be a data set with a final qualifier of C.

Note that the **merge**(*dsname*) form is used only with the OMD370 CLIST. When the **merge** option is used with the LC370 CLIST, no data set name is specified because the location of the source is always the data set name immediately following the command name on the command line, that is, the source code to be compiled.

The following indicates that source code is not to be merged into the OMD's output listing:

```
nomerge
```

When the LC370 CLIST is run, the default is **merge**. When OMD370 is used, the default depends on how the object data set name is specified. If the object data set name is specified in single quotes, the default is **nomerge**. Otherwise, the default is **merge**. (The source data set name is determined by replacing the final OBJ qualifier in the source data set name with C.)

Under CMS, the default is **merge**. By default, source code is merged with the object code in the OMD output listing.

-mrc (USS only)
requests that the compiler return the same return codes as on OS/390 and CMS, that is, 4 if there were warnings, 8 if there were errors, and so on. If **-mrc** is not specified, the compiler conforms to UNIX conventions and returns 0 if there were no errors and a non-zero code if errors were detected.

object (**-o** *filename* under USS)
outputs object code.

In TSO, this option is specified as follows:

```
object(dsname)
```

where *dsname* names the data set into which the compiler stores the object code. The data set name can be a sequential data set or a partitioned data set member. If the data set belongs to another user, the fully qualified name of the data set must be specified, and the name must be preceded and followed by three single quotes, as in the following example:

```
LC370 PROJ4(PART1)
OB(’’’YOURLOG.PROJ4.OBJ(PART1)’’’)
```

The extra quotes are required for the CLIST language. If the data set name is not specified within three single quotes, it is assumed to be a data set with a final qualifier of OBJ.

The following indicates that no object code is to be stored by the compilation:

```
noobject
```

When the **object** or **noobject** option is missing, the default depends on how the source data set name is specified. If the source data set name is specified in single quotes, the default is **noobject**. Otherwise, the default is **object**. (The object data set name is determined by replacing the final C in the source data set name with OBJ.)

In TSO, if both **noobject** and **omd** are specified, object code is generated but discarded after the OMD is run.

Under CMS, the default is **object**. By default, object code is generated in pass two of the compiler. If you specify **noobject**, pass two is suppressed and object code is not generated.

Under CMS, if both **noobject** and **omd** are specified, neither pass two nor the OMD is run.

Under USS, the **-o** option is used to specify the output file for **sascc370**. If the **-c** option is also specified, the **-o** option specifies the file where the compiler’s output (an object module) is to be stored. If **-c** is not specified, the **-o** option specifies the file where the linkage editor’s output (a load module or program object) is to be stored. Use of **-o** where **-c** is not specified is discussed in more detail in “COOL Options” on page 149 under the **load** option.

When **-o** is specified, the syntax is as follows:

```
-o filename
```

The filename is assumed to be an absolute or relative HFS pathname. To store the compiler output in an OS/390 data set, you must use a pathname beginning with two slashes possibly followed by a SAS/C style prefix. (See SAS/C Library Reference, Volume 1 for information about style prefixes.) For example, to store the object file in the PDS member YOURLOG.PROJ4.OBJ(PART1), specify the following:

```
-o '//dsn:yourlog.proj4.obj(part1)'
```

Note that you should not specify an explicit object filename if you compile more than one source file at a time, since each individual compilation will overwrite the object file.

Under the USS shell, if the options **-o** and **-c** are not both specified, the compiler stores its object code output in a default location. If the compiler input file is an HFS file, the object filename is the same as the source filename with the

extension changed to **.o**. If the compiler input file is an OS/390 data set, the object file is an HFS file whose name is derived from the member name, if the input file is a PDS member and, otherwise, from the next-to-the-last qualifier of the data set name. For instance, if you compile the source file **//tso:proj4.c(part1)**, the default object location is the HFS file **part1.o**. If you compile the sequential file **//tso:proj4.report.c**, the default object location is the HFS file **report.o**.

Note that unless **-c** is specified, you cannot override the location where the object module is stored.

omd (**-Komd[=***filename***]** or **-S** under USS)
invokes the object module disassembler (OMD) after successful compilation. OMD-only options and selected compilation options are passed to the OMD, as explained in "Object Module Disassembler Options" on page 105.

Under USS, you can supply the name of the OMD's output file by specifying **-Komd=***filename*. If **-Komd** is specified without a filename, the listing is stored in an HFS file with a **.omd** extension. The name of the default OMD output file is derived from the source filename in the same way as the object file, except for the **.omd** extension. See the **object** option for a description of this process. Note that you should not specify an explicit filename if you compile more than one source file at a time, since each individual compilation will overwrite the OMD output file.

optimize (**-Koptimize** under USS)
causes optimized code to be generated. See "The optimize Option" on page 63 for details on this option.

options (**-Koptions** under USS)
generates an options listing. The options listing contains all options in effect for the compilation.

overstrike (**-Koverstrike** under USS)
prints special characters in the listing file as overstrikes. This option is useful, for example, if you do not have a printer that can print the special characters listed in the fourth column of Table 2.2 on page 22.

pagesize (**-Kpagesize=***nn* under USS)
defines the number of lines per page of source and cross-reference listings. (See Table 6.6 on page 126.) **pagesize** is specified as follows:

- under OS/390 batch: **pagesize(*nn*)**
- in TSO: **pagesize(*nn*)**
- under CMS: **pagesize *nn***

nn lines per page of listing are printed at the location determined by the **print** option. The default is 60 lines per page. (The default location is different for each operating system and is described in the discussion of **print**.)

pflocal (**-Kpflocal** under USS)
assumes that all functions are **__local** unless **__remote** was explicitly specified in the declaration. The default is **nopflocal**. The **nopflocal** option causes the compiler to treat all function pointers as **__remote** unless they are explicitly declared with the **__local** keyword.

posix (**-Kposix** under USS)
informs the compiler that the program is a POSIX oriented program, and that compile-time and run-time defaults should be changed for maximum POSIX

compatibility. The default is **noposix** under TSO, CMS, and OS/390 batch and **-Kposix** under USS.

Specifically, the **posix** option has the following effects on compilation:

- The SAS/C feature test macro **_SASC_POSIX_SOURCE** is automatically defined.
- The compiler option **refdef** is assumed if **norefdef** is not also specified.
- The special POSIX symbols **environ** and **tzname** are automatically treated as **__rent** unless declared as **__norent**.

Additionally, if any compilation in a program's main load module is compiled with the **posix** option, it will have the following effects on the execution of the program:

- The **fopen** function assumes at runtime that all filenames are HFS filenames unless prefixed by **"//"**.
- The system function assumes at runtime that the command string is a shell command unless prefixed by **"//"**.
- The **tmpfile** and **tmpnam** functions refer to HFS files in the **/tmp** directory.

Note: Functions that can be used by both POSIX and non-POSIX applications should be compiled without use of the **posix** compiler option. △

ppix (**-Kppix** under USS)
allows nonstandard use of the preprocessor.

If the **ppix** option is in effect, the preprocessor allows token-pasting by treating a comment in macro replacement text as having zero characters. The ISO/ANSI Standard defines the double pound sign (**##**) operator to perform token-pasting.

This option also specifies that the preprocessor should replace macro arguments in string literals. Equivalent functionality can be gained for portability by using the ISO/ANSI Standard pound sign (**#**) operator.

pponly (**-P** under USS)
creates a file containing preprocessed source code for this compilation. Preprocessed source code has all macros and **#include** files expanded. If the **pponly** option is used, all syntax checking (except in preprocessor directives) is suppressed, no listing file is produced, and no object code is generated.

In TSO, use the following:

```
pponly(dsname)
```

where *dsname* indicates the name of a data set in which the preprocessed source file is to be stored. If the library belongs to another user, the fully qualified name of the data set must be used, and the name must be preceded and followed by three single quotes because of the CLIST language requirements. No final qualifier is assumed for a **pponly** data set.

Under CMS, use **pponly**. The output file is written to a file with the same filename as the source file and a filetype of PP.

Under OS/390 batch, use **pponly**. The output file is written to the data set allocated to the DDname SYSPPOUT.

In TSO and under OS/390 batch, the output data set should use the DCB options LRECL=1028, RECFM=VB. The data set can have any block size.

Under the USS shell, if the **-o** option is specified together with **-P**, the preprocessed source code is written to the file specified by **-o**. If **-o** is not specified, the preprocessed source code is written to an HFS file with a **.i** extension. The name of the default ouput file is derived from the source filename in the same way as the object file, except for the **.i** extension. See the **object** option for a description of this process.

pr
under CMS, specifies the output ASM file for the object module disassembler. The form of the **pr** option is as follows:

```
pr(asm-fileid)
```

where *asm-fileid* is a CMS fileid or a SAS/C **sf:** style filename. The default filetype is ASM. See "Using the OMD under CMS" on page 96.

print (**-Klisting[=***filename***]** under USS)
produces a listing file.

Under OS/390 batch, the **print** option produces a listing file and sends it to SYSPRINT. The listing file also includes error messages. If **noprint** is used, the listing file is suppressed. Under OS/390 batch, the default is **print**.

Also see the discussion of **term**. Table 6.7 on page 127 summarizes the interaction between **term** and **print**.

In TSO, the **print** option is used with both the LC370 CLIST and the OMD370 CLIST to specify where the listing file is to be stored.

If you specify the following, the listing file is printed at the terminal:

```
print(*)
```

If you use **print(*)**, you do not need to use the **term** option. If you do, error messages are sent to the terminal twice. See also **term**.

The following stores the listing file 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 specified, and the name must be preceded and followed by three single quotes because of the CLIST language requirements. If the data set name is not specified within three single quotes, it is assumed to be a data set with a final qualifier of LIST.

The following form specifies that no listing file is to be produced:

```
noprint
```

If you use **noprint**, the compiler ignores all other listing options, such as **pagesize** and **ilist**. The **xref** option also is ignored.

If the source data set name is enclosed in single quotes, the default is **noprint**. Otherwise, the default is **print**. The listing data set name is determined by replacing the final C in the source data set name with LIST and ignoring any member name specification.

You cannot specify **noprint** when you use the OMD370 CLIST.

If you do not specify **print** when you use the OMD370 CLIST, the default is **print(*)** if the object data set name is enclosed by single quotes. Otherwise, the listing data set name is determined by replacing the final OBJ qualifier in the source data set name with LIST, and any member name specification is ignored.

Under CMS, **print** spools the listing file to the printer. **noprint** suppresses the listing file. **noprint** is an alternative to **print**, **disk**, and **type**.

You can also give the **print**, **disk**, and **type** options to the OMD370 EXEC. If you use more than one of the options **type**, **print**, or **disk**, the last one entered is in effect. See Table 6.7 on page 127.

Under USS, by default, no listing file is generated unless you specify the **-Klisting** option. You can supply the name of the listing file by specifying **-Klisting=***filename*. If **-Klisting** is specified without a filename, the listing is stored in an HFS file with a **.lst** extension. The name of the default listing file is derived from the source filename in the same way as the object file, except for the

.lst extension. See the **object** option for a description of this process. Note that you should not specify an explicit filename if you compile more than one source file at a time, since each individual compilation will overwrite the listing file.

rdepth (**-Krdepth=***n* under USS)
defines the maximum level of recursion to be inlined (the default is 1). **rdepth** is used with **optimize** only. See "The optimize Option" on page 63 for more details. **rdepth** is specified as follows:

- under OS/390 batch: **rdepth(*n*)**
- under TSO: **rdepth(*n*)**
- under CMS: **rdepth *n***

redef (**-Kredef** under USS)
allows redefinition and stacking of **#define** names.

refdef (**-Krefdef** under USS)
The **refdef** option forces the use of the strict reference-definition (ref/def) model for external linkage of **__rent** identifiers. If you specify **norefdef**, which is the default, the compiler uses the common model. The minimum abbreviation of **refdef** is **ref**. This option is useful primarily when used with the **rent** or **rentext** options. (Strict reference-definition is always used for **__norent** identifiers.)

Note: If you specify the **posix** option, the compiler option **refdef** is assumed if **norefdef** is not also specified. △

rent (**-Krent** under USS)
specifies that all **extern** and **static** variables are **__rent** by defualt.

rentext (**-Krentext** under USS)
specifies that all **extern** variables are **__rent** by default, and all **static** variables are **__norent** by default.

reqproto (**-cf** under USS)
requires that all functions and function pointers have a prototype in scope. If the **reqproto** option is used and a function or function pointer is declared or defined that does not have a prototype, the compiler issues a warning message.

smpxivec (**-Ksmpxivec** under USS)
causes the compiler to generate a CSECT with a unique name in the place of @EXTERN#. The option is provided to accommodate SMP update methods. Refer to *SAS Programmer's Report: SMP Packaging for SAS/C Based Products* for more information on this option.

sname (**-Ksname=***name* under USS)
defines the section name. The *name* can be up to seven characters in length.
The section name is assigned by the compiler as follows:

- The section name is the name specified by the user with the **sname** option.
- In the absence of a specific compile-time **sname** option, the section name is the name of the first external function in the module, truncated to seven characters.
- If no name is specified with the **sname** option and there is no external function in the module, the section name is the name of the first external variable in the function.
- If no name is specified with the **sname** option, there is no external function in the module, and there is no external variable in the module (that is, the module contains only static data or functions, or both), then the section name is the name @ISOL@.

The following are further instructions for using **sname** in different environments:

- Under OS/390 batch, the specification is as follows:

```
sname(name )
```

where *name* defines the section name.

- In TSO, the specification is as follows:

```
sname(name )
```

- Under CMS, the specification is as follows:

```
sname name
```

source (**-Ksource** under USS)
outputs a formatted source listing of the program to the listing file. (The default location of the listing file is different for each operating system and is described in the discussion of **print**.)

nosource suppresses only the source listing; the cross-reference listing is still printed if requested with the **xref** option.

The **source** option has no effect on the OMD listing if an OMD listing is requested. Whether source code is merged into the OMD listing is controlled by the **merge** option.

stmap (**-Kstmap** under UNIX System Services)
requests that a map of structure elements and their offsets be generated in the cross-reference for each structure tag enclosed. Specifying the **stmap** option implies the **xref** option.

strict (**-Kstrict** under USS)
enables an extra set of warning messages for questionable or nonportable code.

stringdup (**-Kstringdup** under USS)
creates a single copy of identical string constants.

suppress *n* (**-w***n* under USS)
ignores one or more warning conditions. For more information about related messages, see SAS/C Software Diagnostic Messages.

Each warning condition is identified by its associated message number, *n*. Only warnings in the ranges 0-199 and 300-499 are affected. Conditions whose numbers have been specified are suppressed. No message is generated, and the compiler return code is changed.

The following are further instructions for using **suppress** in different environments:

- In TSO, specify the **suppress** option as follows:

```
suppress(n)
```

where *n* is the number of the message associated with the warning condition. If more than one warning condition is to be suppressed, specify the numbers in a comma-delimited list, enclosed by quotes, as follows:

```
suppress('n1,n2,...')
```

- Under CMS, use the following:

```
suppress n
```

or

```
suppress n1 n2 ...
```

- Under OS/390 batch, use the following:

```
suppress(n)
```

or

```
suppress(n1,n2 ...)
```

- Under the USS shell, use the following:

```
-wn
```

or

```
-wn1 -wn2 ...
```

Any number of warning conditions can be specified, regardless of the environment. If both **suppress** and **enforce** specify the same message number, the warning is enforced.

-temp=*directory* (USS only)
specifies a directory where temporary files created by **sascc370** should be stored.

term
directs diagnostic messages to **stderr**.

In contrast to **print**, **term** specifies whether error messages are written to **stderr** but does not affect the contents of any listing file.

Under OS/390 batch, CMS, and TSO, **stderr** is defined as follows:

- under OS/390 batch: the DDname SYSTERM
- in TSO: interactive terminal
- under CMS: interactive terminal

The **term** option interacts with the **print** option as summarized in Table 6.7 on page 127 for OS/390 batch and TSO and in Table 6.8 on page 127 for CMS. Under the USS shell, diagnostic messages are always sent to **stderr**.

trans (**-Ktrans** under USS)
translates special characters to their listing file representations. Default representations for these characters are in column four of Table 2.2 on page 22. If you specify **notrans**, all special characters are written out as they appear in the source data.

trigraphs (**-Ktrigraphs** under USS)
enables translation of ANSI Standard trigraphs. If the **trigraphs** compiler option is used, all occurrences of the following three-character sequences are replaced with the corresponding single character:

```
??=       #
??(       [
??/       \
??)       ]
??<       {
??>       }
??'       ^
??!       |
??-       ~
```

Unlike digraphs, trigraphs are replaced within comments and character string literals. (Digraphs are shown in Table 2.1 on page 20.)

type
displays the listing file on the terminal. **type** is a CMS option only. **type** implies **noterm**.

Note that you cannot use **type** with either **print** or **disk**. If you specify more than one of the options **type**, **print**, or **disk**, the last one you enter is in effect. See also **print**, **term**, and **disk**.

undef (**-Kundef** under USS)
undefines predefined macros.

Predefined macros are defined as follows:

- under OS/390 batch:

```
#define DEBUG 1
#define NDEBUG  1
#define __I370__ 1
#define OSVS 1
```

- in TSO:

```
#define DEBUG 1
#define NDEBUG  1
#define __I370__ 1
#define OSVS 1
```

- under CMS:

```
#define DEBUG 1
#define NDEBUG  1
#define __I370__ 1
#define CMS 1
```

The definition of the **DEBUG** or **NDEBUG** macro depends on whether or not you have specified the **debug** or **nodebug** option.

upper (**-Kupper** under USS)
outputs all lowercase characters as uppercase in the listing file. **upper** implies **overstrike**.

usearch (**-Kusearch** under USS)
specifies that UNIX oriented search rules should be used when the compiler searches for include files rather than mainframe-oriented search rules. This option may be useful when compiling programs ported from a UNIX environment. The effect of **usearch** is described in detail in “Complete include processing” on page 16. **-Kusearch** is the default when compiling under the USS shell, while **nousearch** is the default in all other environments.

-v
specifies verbose mode. In verbose mode, the command line that executes each phase of the compiler is displayed. **-v** is valid for UNIX System Services (USS) only.

verbose (**-v** under USS)
prints relocation directory and line number and offset tables separately, in addition to merging them with the generated code. The **verbose** option applies only to the OMD listing. OMD370 displays the run-time constants CSECT if the **verbose** option is specified. The OMD370 utility also displays the extended names CSECTs when the **verbose** option is specified.

vstring (**-Kvstring** under USS)
generates character string literals with a 2-byte length prefix. This option is used primarily in conjunction with the interlanguage communication feature. For more information on the **vstring** option, refer to Chapter 3, "Communication with Other Languages," in the SAS/C Compiler Interlanguage Communication Feature User’s Guide.

warn (**-Kwarn** under USS)
causes compilation warning messages to be printed. **nowarn** suppresses warning messages.

xref (**-Kxref** under USS)
produces a cross-reference listing.

zapmin (**-Kzapmin=***n* under USS)
specifies the minimum size of the patch area, in bytes. In TSO and under OS/390, use the following:

```
zapmin(n)
```

where *n* refers to the number of bytes in the patch area. The default is 24 bytes.
Under CMS, use the following:

```
zapmin n
```

where *n* refers to the number of bytes in the patch area. The default is 24 bytes.
For more information about the patch area, refer to "Register Conventions and Patch Writing" on page 60. For more information about using the **zapmin** option, refer to "The zapmin option" on page 61.

zapspace (**-Kzapspace=***n* under USS)
changes the size of the patch area generated by the compiler. Under OS/390 batch and in TSO, use the following:

```
zapspace(factor)
```

Under CMS, use the following:

```
zapspace factor
```

For more information about the patch area, refer to "Register Conventions and Patch Writing" on page 60. For more information about using the **zapspace** option, refer to "The zapspace option" on page 61.

Listing File Description

Listing file refers to the file that contains one or more of the types of listings summarized in Table 6.6 on page 126. The contents of the listing file depend on the option or options specified. The first column of Table 6.6 on page 126 shows options that produce different types of listings. The type of listing provided by each option is in the second column.

Table 6.6 Listing File Contents by Option

Option	Type of Listing Written to Listing File
options	options listing.
source	formatted source listing. The formatted source listing may be interspersed with error messages.
xref	cross-reference listing.
omd	object module disassembler listing.

As an example, if you specify **source** and **xref** on the command line when you run the compiler, a formatted source listing and a cross-reference listing are sent to the

listing file. The destination of the listing file is system-dependent and is described with the **print** option (for OS/390 batch, TSO, and CMS) and the **disk** and **type** options (for CMS) in "Option Descriptions" on page 105.

Interaction between the term, print, disk, and type Options

The **term**, **print**, **disk**, and **type** options interact as shown in the following tables.

Table 6.7 Interaction between term and print under OS/390 Batch and in TSO

Option or Options Requested	Result
term, **print**	A listing file (including any error messages) is generated.[1] Error messages also go to **stderr**.[2]
noterm, **print**	A listing file is not generated. Because **noterm** is explicitly requested, error messages are not sent to the terminal.
print (default is **noterm**)	A listing file (including any error messages) is generated. Error messages are not sent to the terminal.
noprint (default is **term**)	No listing file is generated. All error messages go to **stderr**.

1 See the discussion of **print** for the destination of the listing file for your environment.
2 Note that even when **term** is in effect, compiler information banners are not written to **stderr**.

Table 6.8 Interaction between term, print, disk, and type under CMS

Option or Options Requested	Result
term (default is **disk**)	A listing file (including error messages is generated and sent to the A-disk. Messages also go to the terminal.
noterm (default is **disk**)	A listing file is generated. Messages are not sent to the terminal.
disk, **print** (default is **term**)	A listing file (including error messages) is generated. Messages also go to the terminal.
type (default is **term**)	A listing file (including any error messages) is produced and sent to the terminal. Only one copy of messages is sent to the terminal.

Option or Options Requested	Result
noprint (default is **term**)	No listing file is produced. By default, all messages go to the terminal.
noprint, **noterm**	A listing file is not produced. Because **noterm** is explicitly requested, messages are not sent to the terminal.

Preprocessor Options Processing

Preprocessor Symbols

The compiler creates preprocessor symbols for a number of compiler options. The compiler assigns the preprocessor symbol's value to correspond to each option's state. Table 6.9 on page 128 lists the options and symbols, along with their corresponding values that the preprocessor creates.

Table 6.9 Preprocessor Symbols

Option	Symbol	Value
armode	**_O_ARMODE**	1
noarmode	**_O_ARMODE**	0
at	**_O_AT**	1
noat	**_O_AT**	0
bitfield(1)	**_O_BITFIELD**	1
bitfield(2)	**_O_BITFIELD**	2
bitfield(4)	**_O_BITFIELD**	4
nobitfield	**_O_BITFIELD**	0
bytealign	**_O_BYTEALIGN**	1
nobytealign	**_O_BYTEALIGN**	0
comnest	**_O_COMNEST**	1
nocomnest	**_O_COMNEST**	0
cxx	**_O_CXX**	1
nocxx	**_O_CXX**	0
dbhook	**_O_DBHOOK**	1
nodbhook	**_O_DBHOOK**	0
debug	**_O_DEBUG**	1
nodebug	**_O_DEBUG**	0
dollars	**_O_DOLLARS**	1
nodollars	**_O_DOLLARS**	0

Option	Symbol	Value
indep	**_O_INDEP**	1
noindep	**_O_INDEP**	0
inline	**_O_INLINE**	1
noinline	**_O_INLINE**	0
japan	**_O_JAPAN**	1
nojapan	**_O_JAPAN**	0
pflocal	**_O_PFLOCAL**	1
nopflocal	**_O_PFLOCAL**	0
posix	**_O_POSIX**	1
noposix	**_O_POSIX**	0
ppix	**_O_PPIX**	1
noppix	**_O_PPIX**	0
rent	**_O_RENT**	1
norent	**_O_RENT**	0
rentext	**_O_RENTEXT**	1
norentext	**_O_RENTEXT**	0
sname	**_O_SNAME**	value of option
stringdup	**_O_STRINGDUP**	1
nostringdup	**_O_STRINGDUP**	0
trigraphs	**_O_TRIGRAPHS**	1
vstring	**_O_VSTRING**	1
novstring	**_O_VSTRING**	0
zapmin	**_O_ZAPMIN**	value of option
zapspace	**_O_ZAPSPACE**	value of option

The compiler assigns to the **_O_SNAME** symbol the value of the **sname** option, surrounded by quotes. For example, specifying the **sname** option as MYPROG is equivalent to the following preprocessor definition:

```
#define _O_SNAME "MYPROG"
```

If the **sname** option has not been specified, the value assigned to **_O_SNAME** is "".

The **_O_ZAPSPACE** preprocessor symbol is assigned the value of the **zapspace** option. If the **zapspace** option has not been specified, **_O_ZAPSPACE** is assigned a value of 1.

The **_O_ZAPMIN** preprocessor symbol is assigned the value of the **zapmin** option. If the **zapmin** option has not been specified, **_O_ZAPMIN** is assigned a value of 1.

The #pragma options statement

The **#pragma options** statement specifies compiler options within program source code. More than one **#pragma options** statement can be used in a source file. The format of the **#pragma options** statement is as follows:

```
#pragma options copts(option-1,option-2(n))
```

where *option-1* and *option-2* are compiler options, and *n* is the value that an option takes.

For example, you can specify the **bitfield** compiler option in the following manner:

```
#pragma options copts(bitfield(2))
```

where **2** is the value of the **bitfield** option.

Separate multiple options with commas or blanks. Both of the following examples are correct:

- **#pragma options copts (bitfield(2),pagesize(60),dollars)**
- **#pragma options copts (bitfield(2) pagesize(60) dollars)**

The following options can be specified in a **#pragma options** statement:

at	**hlist**	**ppix**
bitfield	**hmulti**	**redef**
comnest	**hxref**	**reqproto**
ctsup	**ilist**	**source**
cwsup	**imulti**	**strict**
dbgmacro	**ixref**	**suppress**
dollars	**maclist**	**trigraphs**
dynamndef	**mention**	**undef**
enforce	**mlist**	**warn**
exclude	**pagesize**	**xref**

Only these options and their negations can be specified, and they must be specified entirely in lowercase and unabbreviated.

There are two other uses of the **#pragma options** statement:

- The **#pragma options push copts** statement pushes the current setting of compiler options to the top of the stack.
- The **#pragma options pop copts** statement returns the compiler options to their values at the time of the last **push** statement.

The following example suppresses the source listing and then returns it to its previous state:

```
#pragma options push copts

#pragma options copts(nosource)

   C statements

#pragma options pop copts
```

The **pragma options push copts** statement saves the current value of the options. **#pragma options copts (nosource)** temporarily suppresses the source listing. The **nosource** option remains in effect until the **#pragma options pop copts** statement appears in the program. The **#pragma options pop copts** statement returns the source listing (and all other options) to their states preceding the **push** statement. If the source listing was suppressed before the **push** statement, it will continue to be suppressed.

CHAPTER

7

Linking C Programs

Introduction

This chapter describes how to link the C programs that you have compiled. First, there is a discussion of the COOL object code preprocessor. Following this discussion are sections that describe linking procedures for each host operating system. Lists and descriptions of COOL options, keywords, and control statements are at the end of the chapter. Other topics discussed include the use of AR370 archives, specifying a program entry point and the names of the various SAS/C Libraries.

The COOL Object Code Preprocessor

COOL (C object-oriented linker) is a utility program that assists in the link-editing of C programs. COOL merges CSECTs for external or static variables; the IBM linkage editor does not have this capability. When the **rent** or **rentext** compiler option is used, the compiler creates a separate control section (CSECT) to contain the external variable initialization data (see "Compiler-generated Names" on page 53) for the current compilation. Data that are to be used for the initialization of external variables are copied during program startup from these CSECTs to dynamically allocated memory. This copy process is necessary to support reentrant execution. If the **rentext** option is used, only initialization data for external variables is stored in the initialization data CSECT. Also, with any compiler option, initialization data for **__rent** variables is stored in the initialization data CSECT. (If a compilation contains no initializations of any of the types described above, no initialization CSECT is created.)

If more than one compilation initializes applicable variables, then all of the initialization CSECTs must be merged before the program can be linked. If they are not combined, the linkage editor ignores all but the first compilation's data since they all have the same CSECT name. Therefore, some initializations would be skipped during execution, with unpredictable results.

COOL merges this initialization data by combining all of the object code for a given program in a manner similar to the CMS loader or OS linkage editor. If any of the object modules contain an initialization CSECT, COOL retains the initialization data and then deletes the CSECT from the object module. When all of the object modules are processed, COOL produces a single object module containing a single merged initialization CSECT, followed by the preprocessed object files.

COOL also checks for **__rent** variables with multiple initial values during the merge. COOL issues a warning for external variables that have multiple initial values. (See "Reentrant and Non-reentrant Identifiers" on page 57 for more information about external variables.)

COOL also prepares object files for linking when the **extname** compiler option is specified. Under the **extname** option, the compiler creates special data objects in the object file that contain the original C identifiers and their associated short forms. COOL reads these data objects and then creates unique external symbols in the output

object file, thus enabling the linkage editor or loader to properly link the output object file by using these unique external symbols. Refer to "Linking Programs under CMS" on page 134 for more information about extended names processing.

Under CMS, C programs compiled with the **rent** or **rentext** option may exceed the LOAD pseudoregister limit. By default, COOL changes the pseudoregister definitions in the object modules to a form that the LOAD command can handle. For more information on COOL and the CMS LOAD command, refer to "Linking Programs under CMS" on page 134.

Prior to Release 6.50, a problem frequently encountered was an attempt to process an object deck with COOL that had already been prelinked by COOL. This caused a number of problems, not obviously related to the attempt to reprocess an object with COOL, and usually resulted in an ABEND. In this release, COOL marks each object deck as it is processed and if an attempt is made to reprocess the marked object, COOL produces a diagnostic message indicating the condition.

The new processing is divided into two phases. The first phase marks the output object deck to indicate it has already been processed with COOL. It is controlled by the **allowrecool** and **noallowrecool** options. The second phase detects that an input object deck has been marked to indicate it was previously processed. The second phase is controlled by the **ignorerecool** and **noignorerecool** options. By default, COOL marks the object deck to prevent an attempt to reprocess it. Also by default, COOL detects that the input object deck was previously processed by COOL.

Note: These defaults can cause COOL to indicate an error where it would not detect such an error in previous releases. Under certain restricted circumstances, it is possible to generate object code that can be successfully processed by COOL more than once. If this behavior is desired, the options can be specified such that the output object's decks are not marked and/or that such marking be ignored. △

Note: You may not reprocess COOL output with COOL if any object file was compiled with the **extname** option. △

When to Use COOL

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

- More than one compilation initializes a **__rent** variable. There are four ways a variable is assigned the **__rent** attribute:
  - The variable is external and the compiler option **rent** or **rentext** is used.
  - The variable is static and the compiler option **rent** is used.
  - The variable is external and the name begins with an underscore.
  - The variable is declared **__rent**.

  For more information on the **__rent** attribute see "Reentrant and Non-reentrant Identifiers" on page 57.
- More than one compilation was compiled with the **extname** option.
- Under CMS, the cumulative length of the pseudoregister vector exceeds the maximum size permitted by the loader.
- At least one C++ function is used.
- The SAS/C All-Resident Library was used.
- Some of the object modules are stored in an AR370 archive.

Under OS/390, if you fail to run COOL in a situation that requires it, the linkage editor will generate message IEW0461 or IEW2480W referencing the symbol **NO_CLINK**.

This message indicates that use of COOL was required, and that the resulting load module is unlikely to execute correctly.

Under CMS, if you attempt to load object code that requires the use of COOL and COOL has not been used to preprocess the object modules, the loader issues message DMSLIO020W referencing the symbol **NO_CLINK**. If this situation arises, the use of COOL is required.

A special case

For most programs compiled with the **norent** and **noextname** options, you do not need to use COOL. The only exception to this rule is if more than one compilation initializes external variables that begin with an underscore (such as the **_options** variable). External variables that begin with an underscore are always stored as pseudoregisters, even if the **norent** option is used; therefore, an initialization CSECT is created.

Using COOL to Link Programs

The design of COOL requires that all input to COOL be in the form of object modules, including any automatic call libraries. COOL also accepts control statements similar to those used by the linkage editor (see “COOL Control Statements” on page 161). As stated, the output from COOL is an object module and becomes input to the linkage editor or loader.

COOL processes object modules in the same order and with the same restrictions as imposed by the linkage editor. However, there is no guarantee that unresolved references during autocall processing are reconciled in the same order with COOL as with the linkage editor because the linkage editor may process the corresponding compilations in any order. The only time this could cause a problem is if several members of an autocall library contain copies of the same object module, in which case the copy that is actually used is unpredictable. Proper autocall library management should prevent this situation from occurring.

Linking Multilanguage Programs

Object modules produced by an assembler or another compiler do not need to be preprocessed by COOL. If the C object modules do not require the use of COOL, then the C object modules and non-C object modules can be linked in the normal manner, without using COOL. If, however, the C object modules must be preprocessed by COOL, the non-C object modules should be linked with the COOL output module by the LINK, LOAD, or LKED command in a separate step.

When you use the ILC feature to mix SAS/C code with code in another language, use the ILCLINK utility to produce the module. Refer to Chapter 8, "Linking Multilanguage Programs with the ILCLINK Utility," in the SAS/C Compiler Interlanguage Communication Feature User’s Guide for details.

Linking Programs under CMS

The CMS LOAD command is limited in the number and size of pseudoregisters it can handle. The number and length of the pseudoregisters are directly related to the number and size of external variables in the program. For example, an external **int**

array with 1000 elements causes a pseudoregister to be created that is **1000*sizeof(int)**, or 4000 bytes long. C programs that have been compiled with the **rent** or **rentext** options may, in some cases, produce too many pseudoregisters or the cumulative length of the pseudoregisters may be too large for the LOAD command to process.

The CMS LOAD command cannot process pseudoregisters that have a cumulative length of more than 4K. If this length is exceeded, the LOAD command does not necessarily produce an error message. You can diagnose this situation in a MODULE by examining the LOAD MAP after the GENMOD command has completed. The pseudoregister addresses listed in the VALUE column should always be in increasing order. If they are not, the maximum cumulative length is exceeded.

The maximum number of pseudoregisters that the CMS LOAD command can handle is indeterminate. The LOAD command typically issues the following error message if the number is exceeded:

```
DMSLIO168S: PSEUDO-REGISTER TABLE OVERFLOW
```

COOL performs pseudoregister removal; it does not affect program execution or reentrancy in any way. The **noprem** option suppresses this function. The COOL output file, COOL370 TEXT, may not be reprocessed by COOL unless the **noprem** option has been used.

The COOL EXEC

The COOL EXEC invokes the COOL object code preprocessor and can optionally invoke the CMS LOAD, START, GENMOD, or LKED command. The format is as follows:

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

where filename1, filename2, and so on are the names of the files that are to be the primary input to COOL. Each file should have a filetype of TEXT and contain either object code or COOL/linkage editor control statements. (INCLUDE statements are an example of control statements, as discussed later in this chapter.) 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.

Before invoking COOL, issue the CMS command GLOBAL TXTLIB for any TEXT libraries that COOL should use for autocall resolution. For standard C programs, LC370BAS TXTLIB and LC370STD TXTLIB should be GLOBALed before invoking COOL for any program.

The RESET option

The CMS LOAD command may not select the correct entry point for your program. It is usually best to use the RESET option of the LOAD command to specify the entry point explicitly. If the main function is the C **main** function and you are using the normal C entry point, specify RESET MAIN. If you are using the LKED command to link the program, use the ENTRY control statement to specify MAIN as the entry point.

The CMS GENMOD command may not save all of the CSECTs in the program, especially if you use RESET to specify an entry point. Use the FROM option of the GENMOD command to specify the name of the initial CSECT. This name can be found by reading the LOAD MAP file produced by the LOAD command.

See “Specifying the Correct Entry Point” on page 167 for additional information on defining an entry point.

COOL Listing Output

A number of COOL options, such as **list**, **prmap**, and **enxref**, cause output to be written to the COOL listing file. By default, the COOL listing file is named COOL370 COOLLIST. The **print** option may be used to direct the listing to a different file.

Linking All-Resident Programs

The all-resident library is LCARES TXTLIB. To link an all-resident program as a MODULE, issue the CMS GLOBAL TXTLIB command, naming LCARES before LC370STD and LC370BAS (the normal resident library), and then invoke COOL. For example, suppose a program is made up of three TEXT files, MAINPROG, SUB1, and SUB2, and autocalled routines are in MYLIB TXTLIB. The MODULE can be created with the following commands:

```
GLOBAL TXTLIB MYLIB LC370STD LC370BAS
COOL MAINPROG SUB1 SUB2 (GENMOD TESTPROG
```

Note: See Chapter 10, “All-Resident C Programs,” on page 201 for information on how to modify the source of an existing program to exploit the all-resident library. △

The CMS LOAD command may issue the following message:

```
DMSLIO116S LOADER TABLE OVERFLOW
```

This indicates that there are insufficient virtual machine loader tables to contain the symbols in the TEXT files. Use the CMS command SET LDRTBLS nn, where nn is an integer greater than 3, to define additional loader tables. In order to ensure that the loader tables are allocated successfully, this command should be issued immediately after IPL. Refer to the CMS Command Reference for more information about the SET LDRTBLS command.

Linking Programs in TSO

The following sections describe how to link your C program in TSO.

The COOL CLIST

The COOL CLIST invokes the COOL object code preprocessor, followed by the linkage editor. Optionally, the COOL step can be skipped. The format is as follows:

```
COOL 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. (INCLUDE statements are an example of control statements, as discussed later in this chapter.) Follow standard TSO naming conventions; that is, if the data set belongs to some other user, the full name of the data set must be specified and the name must be enclosed in single quotes. 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 (described in the following section) indicate COOL options, linkage editor options, or the names of other data sets to use during linking.

Executing COOL with the IBM Linkage Editor

COOL accepts the **NOCLINK** option, which causes the linkage editor to be invoked directly without use of the COOL utility.

Linkage editor options

COOL 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 MVS/XA Linkage Editor and Loader User's Guide discusses these options.

Linking All-Resident Programs

When linking an all-resident program, include an object deck created by compiling a source file that includes **<resident.h>** and the appropriate macro definitions. See Chapter 10, "All-Resident C Programs," on page 201 for more information. For example, suppose the PDS member INCNTL contains the following COOL control statements:

```
INCLUDE OBJLIB(MAINPROG)
INCLUDE OBJLIB(SUB1)
INCLUDE OBJLIB(SUB2)
```

The program also autocalls other members from MY.PROG.OBJ. Normally, the COOL command to link this program is

```
COOL PROG(INCNTL) LIB(’’’MY.PROG.OBJ’’’) ...
```

INCNTL would contain the following COOL control statements:

```
INCLUDE OBJLIB(MAINPROG)
INCLUDE OBJLIB(SUB1)
INCLUDE OBJLIB(SUB2)
INCLUDE OBJLIB(RESLIST)
```

The COOL command to link an all-resident version of this program is as follows:

```
COOL PROG(INCNTL) LIB(’’’MY.PROG.OBJ’’’) ALLRESIDENT
```

Linking Programs from the UNIX System Services Shell

Under UNIX System Services (USS), the **sascc370** command is used to link SAS/C programs as well as to compile them. The syntax of **sascc370** is as follows:

```
sascc370 [options] filename1 [filename2 ... ]
```

The *options* argument is a list of compiler options (see Chapter 6, "Compiler Options," on page 101), COOL options, and OS/390 linkage editor options. The filename arguments may specify any combination of C source files, object modules, and AR370 archives. Any input source files are compiled, after which the compiler's output is linked with the object files and the archives. If you call **sascc370** with a list of files which are all object files and archives, the compiler is not invoked. The object files and

archives are passed directly to COOL, and then the output of COOL is passed to the linkage editor.

An object file passed to **sascc370** may also contain COOL control statements. (Note that the file must have a name ending with **.o** for an HFS file or .OBJ for an OS/390 data set.) COOL processes its input files in binary mode. For this reason, an HFS file containing COOL control statements has the following requirements. Each control statement must appear as an 80-byte blank-padded card image, and the control statements must not be separated by new-line characters. One way of creating such an HFS file is to create the control statements in an OS/390 card image data set, and then use the BINARY option of the OCOPY TSO command to copy it to the HFS.

The USS COOL options are described later in "COOL Options" on page 149. To specify OS/390 linkage editor options, the **sascc370 -B** option is used. Multiple **-B** specifications can be used, and each **-B** can specify more than one linkage editor option. For instance, the following command specifies the linkage editor options RENT, LET, and RMODE=24 and stores the output module in the OS/390 PDS *userid*.PROG.LOAD.

```
sascc370 -Brent,let -Brmode=24 -o '//prog.load(app4)' app4.o
```

Note that **sascc370** passes the linkage editor MSGLEVEL=4 unless you specify **-Bmsglevel=**n yourself. This option suppresses linkage editor messages that are not ordinarily wanted. Because this option suppresses the output produced by the linkage editor LIST option, if you specify **-Blist** you should also specify **-Bmsglevel=0**, to allow the LIST messages to be written.

Linking Programs under OS/390 Batch

The following sections discuss the cataloged procedures provided for linking C programs. Ask your SAS Software Representative for C compiler products for the appropriate data set names for your site.

Using Cataloged Procedures to Link

Three cataloged procedures are provided for compiling and linking or simply linking a C program. LC370L and LC370CL should be used to link programs that do not require the use of COOL (non-reentrant programs that do not use **extname**, AR370 archives, or the all-resident library). LC370L and LC370CL do not invoke COOL. LC370LR should be used to link programs that require the use of COOL. This procedure runs COOL before invoking the linkage editor.

The resident library data sets are provided in both object module and load module formats. Those procedures, such as LC370L, which run the linkage editor directly use the load module format data sets. The procedures, such as LC370LR, which run COOL and, then the linkage editor, use the object module format data sets. Note that if you are running COOL, you cannot concatenate libraries in load module format to SYSLIB. The DDname SYSLDLIB should be used for load module format libraries to be accessed by the linkage editor when running COOL.

Link-Editing without COOL

The LC370L and LC370CL cataloged procedures are used to link-edit C programs that do not require COOL. Both procedures link the program with the load module form of the resident library.

Selecting the entry point

If your program requires an entry point other than the standard MAIN entry, the entry point must be explicitly specified. See "Specifying the Correct Entry Point" on page 167 for further information.

Selecting the program environment

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

```
ENV=STD ENV=SPE
```

The default is ENV=STD, which specifies the standard OS/390 environment. ENV=SPE specifies a program that uses the minimal SPE environment. See Chapter 14, "Systems Programming with the SAS/C Compiler," on page 273 for more information about SPE.

The LC370L Cataloged Procedure

Typical JCL for running the cataloged procedure LC370L to link-edit a procedure is shown in Example Code 7.1 on page 139. Both the LC370L and the LC370CL cataloged procedure listings follow the sample JCL.

Example Code 7.1 Sample JCL for Link-Editing with Procedure LC370L

```
//JOBNAME  JOB job card information
//*---------------------------------------------------------------
//*   LINK EDIT A C PROGRAM
//*---------------------------------------------------------------
//LINK      EXEC  LC370L,PARM.LKED='options'
//*---------------------------------------------------------------
//*   REPLACE GENERIC NAMES AS APPROPRIATE
//*---------------------------------------------------------------
//LKED.SYSLMOD  DD  DISP=SHR,DSN=your.load.library(member)
//LKED.SYSIN    DD  DSN=your.object.library(member),DISP=SHR
//LKED.libname  DD  DSN=your.object.library ,DISP=SHR
//
```

The LKED.*libname* DD statement is required if you use the linkage editor INCLUDE *libname* control statement. SYSIN can be a file of object code or control statements. (See the IBM OS/390 linkage editor and loader documentation for your particular installation.) Any linkage editor options can go in the PARM.LKED string. If no options are provided, LIST and MAP are assumed. The LC370L procedure contains the JCL shown in Example Code 7.2 on page 139.

Example Code 7.2 Expanded JCL for LC370L

```
//LC370L   PROC ENTRY=MAIN,ENV=STD,
//         CALLLIB='SASC.BASELIB',
//         SYSLIB='SASC.BASELIB'
//*************************************************************
//*  NAME:  LC370L                              (LC370L)       ***
//*  PROCEDURE:  LINKAGE                                       ***
```

```
//*  DOCUMENTATION:  SAS/C COMPILER AND LIBRARY USER'S GUIDE   ***
//*  FROM:  SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC     ***
//*************************************************************
//*

//* ************************************************************
//* ENV=STD:     MODULE RUNS IN THE NORMAL C ENVIRONMENT
//* ENV=SPE:     MODULE USES THE SYSTEMS PROGRAMMING ENVIRONMENT
//* ENTRY=MAIN:  MODULE IS A NORMAL C MAIN PROGRAM
//* ENTRY=DYN:   MODULE IS DYNAMICALLY LOADABLE AND REENTRANT
//* ENTRY=DYNNR: MODULE IS DYNAMICALLY LOADABLE AND NON-REENTRANT
//* ENTRY=OS:    MODULE IS AN OS SPE APPLICATION
//* ENTRY=OE:    MODULE IS AN USS SPE APPLICATION
//* ENTRY=NONE:  ENTRY POINT TO BE ASSIGNED BY USER
//* ************************************************************
//LKED     EXEC PGM=LINKEDIT,PARM='LIST,MAP',REGION=1536K
//SYSPRINT DD SYSOUT=A,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSTERM  DD SYSOUT=A
//SYSLIN   DD DSN=C.SASC.BASEOBJ(EP@&ENTRY),
//            DISP=SHR
//         DD DDNAME=SYSIN
//SYSLIB   DD DSN=C.SASC.&ENV.LIB,
//            DISP=SHR                      STDLIB OR SPELIB
//         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))
```

Note the following about this example:

- The symbolic parameter SYSLIB refers to the data set name for the automatic call library. Do not override this parameter.
- The symbolic parameter ENTRY can be changed to DYN, DYNNR, OS, OE, or NONE. Refer to "Specifying the Correct Entry Point" on page 167 for a discussion of these parameters.
- The symbolic parameter ENV= refers to the environment under which the program is to run. ENV=STD is the default and specifies the standard OS/390 environment. ENV=SPE should be used to link an SPE application.
- The symbolic parameter CALLLIB can be used to specify a load module call library to be used in addition to the resident library data sets.

The LC370CL Cataloged Procedure

The LC370CL procedure can be used to compile and link-edit a program that does not require preprocessing by COOL. LC370CL contains the JCL shown in Example Code 7.3 on page 140. This JCL is correct as of the publication of this guide. However, it may be subject to change.

Example Code 7.3 Expanded JCL for LC370CL

```
//LC370CL  PROC ENTRY=MAIN,ENV=STD,
//         CALLLIB='SASC.BASELIB',
```

```
//          MACLIB='SASC.MACLIBC',
//          SYSLIB='SASC.BASELIB'
//*************************************************************
//*  NAME:  LC370CL                              (LC370CL)      ***
//*  SUPPORT:  C COMPILER DIVISION                              ***
//*  PRODUCT:  SAS/C                                            ***
//*  PROCEDURE:  COMPILATION AND LINKAGE                        ***
//*  DOCUMENTATION:  SAS/C COMPILER AND LIBRARY USER'S GUIDE    ***
//*  FROM:  SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC      ***
//*************************************************************
//*
//* ***************************************************************
//* ENV=STD:     MODULE RUNS IN THE NORMAL C ENVIRONMENT
//* ENV=SPE:     MODULE USES THE SYSTEMS PROGRAMMING ENVIRONMENT
//* ENTRY=MAIN:  MODULE IS A NORMAL C MAIN PROGRAM
//* ENTRY=DYN:   MODULE IS DYNAMICALLY LOADABLE AND REENTRANT
//* ENTRY=DYNNR: MODULE IS DYNAMICALLY LOADABLE AND NON-REENTRANT
//* ENTRY=OS:    MODULE IS AN OS SPE APPLICATION
//* ENTRY=OE:    MODULE IS AN USS SPE APPLICATION
//* ENTRY=NONE:  ENTRY POINT TO BE ASSIGNED BY USER
//* ***************************************************************
//C        EXEC PGM=LC370B
//STEPLIB  DD DSN=SASC.LOAD,
//            DISP=SHR        COMPILER LIBRARY
//         DD DSN=SASC.LINKLIB,
//            DISP=SHR       RUNTIME 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,DCB=(RECFM=FB,LRECL=80)
//SYSLIB   DD DSN=&MACLIB,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
//*
//LKED     EXEC PGM=LINKEDIT,PARM='LIST,MAP',COND=(8,LT,C)
//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.BASEOBJ(EP@&ENTRY),
//            DISP=SHR
//         DD DDNAME=SYSIN
//SYSLIB   DD DSN=SASC.&ENV.LIB,
//            DISP=SHR               STDLIB OR SPELIB
//         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))
```

Note the following about this example:

- The symbolic parameter ENTRY can be changed to DYN, DYNNR, OS, OE, or NONE. Refer to “Specifying the Correct Entry Point” on page 167 for a discussion of these parameters.
- The symbolic parameter MACLIB refers to the data set name chosen by your installation for the macro library. The symbolic parameter SYSLIB refers to the data set name for the autocall library. Do not override these parameters.
- When you use LC370CL more than once in a job, provide overriding JCL (DISP=(OLD,PASS)) to reuse the compiler SYSLIN data set (&&OBJECT) in all but the first instance.
- When you override SYSPRINT in the compile step to reference a disk data set, the data set disposition must be MOD. The data set must not be a member of a PDS.
- When overriding the SYSLMOD DD statement, you may want to nullify the SPACE parameter by coding an explicit SPACE= keyword. In the absence of a SPACE= specification, the operating system will use the SPACE=*value* operands specified in the procedure, and the existing space attributes of the user SYSLMOD DD may be modified.

Link-Editing with COOL

The LC370CLR and LC370LR cataloged procedures are used to link a program after invoking the COOL preprocessor. Use of the COOL preprocessor is required for many programs, as described at the start of this chapter. Both procedures link the program with the object module form of the resident library.

The LC370LR Cataloged Procedure

Typical JCL for running the cataloged procedure LC370LR to invoke COOL and link-edit a procedure is shown in Example Code 7.4 on page 142.

Example Code 7.4 Sample JCL for Executing COOL with Procedure LC370LR

```
//JOBNAME  JOB job card information
//*-------------------------------------------------------------
//*   LINK EDIT A C PROGRAM WITH COOL
//*-------------------------------------------------------------
//LINK      EXEC  LC370LR,PARM.LKED='options'
//*-------------------------------------------------------------
//*   REPLACE GENERIC NAMES AS APPROPRIATE
//*-------------------------------------------------------------
//LKED.SYSLMOD  DD  DISP=SHR,DSN=your.load.library(member)
//LKED.SYSIN    DD  DSN=your.object.library(member) ,DISP=SHR
//LKED.libname  DD  DSN=your.object.library ,DISP=SHR
//LKED.SYSLDLIB DD  DSN=your.autocall.load.library ,DISP=SHR
//
```

LC370LR does not support the ENTRY symbolic parameter. However, you need to supply an ENTRY statement when you use LC370LR only if you require an unusual entry point, such as $MAINC, $MAINO, a function compiled with the **indep** option, or a specialized SPE start-up routine. Dynamically loadable modules and SPE modules that use a standard start-up routine should be linked correctly by LC370LR without an explicit ENTRY specification.

The LKED.*libname* DD statement is required if you use the linkage editor INCLUDE *libname* control statement.

SYSIN can be a file of object code or control statements. (See the IBM OS/390 linkage editor and loader documentation for your particular installation.)

Any linkage editor options can go in the PARM.LKED string. If no options are provided, LIST and MAP are assumed. In addition, COOL accepts the options listed in Table 7.4 on page 150.

SYSLDLIB includes any user autocall libraries needed in load module form. References to members of SYSLDLIB are left unresolved by COOL and are resolved by the linkage editor.

The LC370LR procedure contains the JCL shown in Example Code 7.5 on page 143. This JCL is correct as of the publication of this guide. However, it may be subject to change. Note that LC370LR supports ENV=GOS for a program that uses GOS (generalized operating system interface).

Example Code 7.5 Expanded JCL for LC370LR

```
//LC370LR  PROC ENV=STD,ALLRES=NO,
//            CALLLIB='SASC.BASEOBJ',
//            SYSLIB='SASC.BASEOBJ'
//*************************************************************
//*  PRODUCT:  SAS/C                                          ***
//*  PROCEDURE:  COOL LINKAGE EDITOR PREPROCESSOR & LINK EDIT  ***
//*  DOCUMENTATION:  SAS/C COMPILER AND LIBRARY USER'S GUIDE   ***
//*  FROM:  SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC     ***
//*************************************************************
//*
//* ENV=STD:     MODULE RUNS IN THE NORMAL C ENVIRONMENT
//* ENV=CICS:    MODULE RUNS IN A CICS C ENVIRONMENT
//* ENV=GOS:     MODULE RUNS USING THE GENERALIZED OPERATING
//*              SYSTEM INTERFACE
//* ENV=SPE:     MODULE USES THE SYSTEMS PROGRAMMING ENVIRONMENT
//* ********************************************************************
//LKED     EXEC PGM=COOLB,PARM='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=A,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSTERM  DD SYSOUT=A
//SYSLIN   DD UNIT=SYSDA,DSN=&&LKEDIN,SPACE=(3200,(20,20)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSLIB   DD DDNAME=AR#&ALLRES       ARESOBJ OR ENVIRONMNENT 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.ARESOBJ,
```

```
//            DISP=SHR
```

Note: The symbolic parameter SYSLIB refers to the data set name for the automatic call library. Do not override this parameter. △

The LC370CLR Cataloged Procedure

The LC370CLR procedure can be used to compile and link-edit a program, invoking the COOL linkage editor preprocessor during the linkage step. LC370CLR contains the JCL shown in Example Code 7.6 on page 144. This JCL is correct as of the publication of this guide. However, it may be subject to change.

Example Code 7.6 Expanded JCL for LC370CLR

```
//LC370CLR  PROC ENV=STD,ALLRES=NO,
//          CALLLIB='SASC.BASEOBJ',
//           MACLIB='SASC.MACLIBC',
//           SYSLIB='SASC.BASEOBJ'
//*************************************************************
//*  NAME:  LC370CLR                                (LC370CLR)   ***
//*  PRODUCT:  SAS/C                                             ***
//*  PROCEDURE:  COMPILATION, PRELINK (COOL) AND LINKAGE         ***
//*  DOCUMENTATION:  SAS/C COMPILER AND LIBRARY USER'S GUIDE     ***
//*  FROM:  SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC       ***
//*************************************************************
//*
//* ****************************************************************
//* ENV=STD:     MODULE RUNS IN THE NORMAL C ENVIRONMENT
//* ENV=GOS:     MODULE RUNS USING THE GENERALIZED OPERATING
//*              SYSTEM INTERFACE
//* ENV=SPE:     MODULE USES THE SYSTEMS PROGRAMMING ENVIRONMENT
//* ****************************************************************
//C         EXEC PGM=LC370B
//STEPLIB   DD DSN=SASC.LOAD,
//             DISP=SHR        COMPILER LIBRARY
//          DD DSN=SASC.LINKLIB,
//             DISP=SHR       RUNTIME 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,DCB=(RECFM=FB,LRECL=80)
//SYSLIB    DD DSN=&MACLIB,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
//*
//LKED      EXEC PGM=COOLB,PARM='LIST,MAP',COND=(8,LT,C),REGION=1536K
//STEPLIB   DD DSN=SASC.LOAD,
//             DISP=SHR        COMPILER LIBRARY
```

```
//         DD DSN=SASC.LINKLIB,
//            DISP=SHR       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)
//SYSLIB   DD DDNAME=AR#&ALLRES      ARESOBJ OR STDOBJ OR SPEOBJ
//         DD DSN=SASC.&ENV.OBJ,
//            DISP=SHR               STDOBJ OR SPEOBJ
//         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.ARESOBJ,
//            DISP=SHR
//SYSIN    DD DSN=*.C.SYSLIN,DISP=(OLD,PASS),VOL=REF=*.C.SYSLIN
```

Note that the SYSLIB concatenation must contain only object-format data sets. If you need to autocall routines from load module format libraries, you should specify the SYSLDLIB DD statement. Members in SYSLDLIB are left unresolved by COOL and are resolved by the linkage editor.

Linking All-Resident Programs

All-resident programs can be linked with the cataloged procedures that invoke COOL, LC370LR, or LC370LRG. (Note the **ALLRES=YES** symbolic parameter is available for both procedures.)

When linking an all-resident program, use the **ALLRES=YES** symbolic parameter, and include an object deck created by compiling a source file that includes **<resident.h>** and the appropriate macro definitions. See Chapter 10, “All-Resident C Programs,” on page 201 for more information.

For example, suppose a program consists of three object files, MAINPROG, SUB1, and SUB2. The program also includes object code from a PDS named MY.PROG.OBJLIB. The JCL to normally link this program is shown in Example Code 7.7 on page 145.

Example Code 7.7 Sample LC370LR JCL

```
//JOBNAME  JOB job card information
//LINK      EXEC  LC370LR,PARM.LKED='options'
//*
//LKED.SYSLMOD  DD  DISP=SHR,DSN=MY.PROG.LOAD(TESTPROG)
//LKED.SYSIN    DD  DSN=MY.PROG.OBJ(MAINPROG),DISP=SHR
//              DD  DSN=MY.PROG.OBJ(SUB1),DISP=SHR
//              DD  DSN=MY.PROG.OBJ(SUB2),DISP=SHR
//LKED.OBJLIB   DD  DSN=MY.PROG.OBJLIB,DISP=SHR
//
```

To create an all-resident version of the load module, include the object file generated by compiling a C source file containing **<resident.h>** and the appropriate macro

definitions, and add the **ALLRES=YES** JCL parameter, as shown in Example Code 7.8 on page 146.

Example Code 7.8 Sample LC370LR JCL for Linking an All-Resident Program

```
//JOBNAME  JOB job card information
//LINK      EXEC  LC370LR,PARM.LKED='options',ALLRES=YES
//*
//LKED.SYSLMOD  DD  DISP=SHR,DSN=MY.PROG.LOAD(TESTPROG)
//LKED.SYSIN    DD  DSN=MY.PROG.OBJ(MAINPROG),DISP=SHR
//              DD  DSN=MY.PROG.OBJ(SUB1),DISP=SHR
//              DD  DSN=MY.PROG.OBJ(SUB2),DISP=SHR
//              DD  DSN=MY.PROG.OBJ(RESLIST),DISP=SHR
//LKED.OBJLIB   DD  DSN=MY.PROG.OBJLIB,DISP=SHR
//
```

Note: Alternately, to create an all-resident version of the load module, you could add the **#include** and appropriate **#define** statements to an existing source file (for instance, MAINPROG) and recompile that source file. △

COOL and Linkage Editor JCL Requirements

This section discusses the data definition (DD) statements needed to run COOL and the linkage editor, if you are writing your own JCL. COOL, like the compiler and OMD, requires that short-form options be used when COOL is invoked directly.

You need the DD statements shown in Table 7.1 on page 146 to invoke COOL.

Table 7.1 Data Sets Needed for Running COOL

DDname	Contents
STEPLIB	compiler library and transient library (unless already in your system libraries).
SYSIN	your primary input file. This must include all the C bject code not autocalled or COOL control statements (or both) to cause the C object code to be included.
SYSLIB	any SAS/C and user autocall libraries needed, including at least SASC.BASEOBJ. These libraries must be in object format.
SYSLIN	output object data set produced by COOL.
SYSPRINT	standard ouput.
SYSTERM	error output.
libname	user-defined library for external references not in SYSLIB (where *libname* is defined in the COOL INCLUDE statement). *libname* is optional.

You need the DD statements shown in Table 7.2 on page 147 to invoke the linkage editor.

Table 7.2 Data Sets Needed for Running the Linkage Editor

DDname	Contents
SYSUTI	temporary work data set as shown in JCL.
SYSLIN	C object code produced by COOL.
libname	user-defined library for external references not in SYSLIB (where *libname* is defined in the INCLUDE statement). *libname* is optional.
SYSLIN	output object data set produced by COOL.
SYSPRINT	standard ouput.
SYSTERM	error output.
libname	user-defined library for external references not in SYSLIB (where *libname* is defined in the COOL INCLUDE statement). *libname* is optional.
SYSLIB	one or more autocall libraries. (To use more than one, use JCL concatenation.) These libraries can be object code or load libraries, but they must all be the same type. Subroutine libraries for other languages or for application packages like GDDM and ISPF are normally referenced by SYSLIB. This should not contain C code if COOL is used because all C code should be included to COOL.
SYSLMOD	the output load module.
SYSPRINT	message file.

Example Code 7.9 on page 147 shows sample JCL for running COOL and the linkage editor.

Example Code 7.9 Sample JCL for Running COOL and the Linkage Editor

```
//JOBNAME  JOB job card information
//*-------------------------------------------------------------
//*   RUN THE COOL LINKAGE EDITOR PREPROCESSOR
//*-------------------------------------------------------------
//COOL     EXEC  PGM=COOL#,REGION=2048K
//STEPLIB     DD  DISP=SHR,DSN=compiler.loadlib
//            DD  DISP=SHR,DSN=sasc.transient.library
//*-------------------------------------------------------------
//*   REPLACE GENERIC NAMES AS APPROPRIATE
//*-------------------------------------------------------------
//SYSIN    DD  DSN=c.primary.input ,DISP=SHR
//SYSLIB   DD  DSN=sasc.baseobj ,DISP=SHR
//         DD  DSN=sasc.stdobj ,DISP=SHR
//         DD  DSN=c.user.subroutine.library,DISP=OLD
//SYSLIN   DD  DSN=cool.output.object.dataset,DISP=SHR
//libname   DD  DSN=user.defined.object,DISP=SHR
//SYSPRINT DD  SYSOUT=class
```

```
//SYSTERM  DD  SYSOUT=class
//*-------------------------------------------------------------
//*   RUN LINKAGE EDITOR
//*-------------------------------------------------------------
//LINK    EXEC  PGM=IEWL,PARM='LIST,MAP,XREF,LET',COND=(8,LE,COOL)
//SYSUT1    DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLIN    DD DSN=cool.output.object.dataset,DISP=SHR
//libname DD DSN=user.defined.library,DISP=SHR
//SYSLIB    DD DSN=your.autocall.library,DISP=SHR
//SYSLMOD   DD DISP=SHR,DSN=output.load.library(member)
//SYSPRINT  DD SYSOUT=class
//
```

Linking USS Programs

It is possible to use a batch cataloged procedure such as LC370L or LC370LR to generate an executable program stored in the USS hierarchical file system. To do so, you must override the SYSLMOD DD statement, with a DD statement of the form:

```
//LKED.SYSLMOD DD PATH='output-module-path',
// PATHOPTS=(OWRONLY,OCREAT),
// PATHMODE=(SIRWXU)
```

Note that output-module-path should be replaced by the full pathname where the executable module is to be stored. The name can be at most 255 characters. Also note that you may wish to use a different PATHMODE specification. The specification shown restricts access to the output module to the user who creates it.

COOL Options (Short Forms)

COOL takes short-form options, as summarized in Table 7.3 on page 148. The options can be in upper- or lowercase. As with the short forms of the compiler and OMD options, you specify positive forms of the option with a hyphen (–) as the initial character; specify negative forms with an exclamation point (!) or not sign (¬). Refer to "COOL Options" on page 149 for complete descriptions of these options.

Table 7.3 COOL Options Equivalents

Long Form	Short Form
auto	**---a**
clet	**---m**
clet(all)	**---m**
clet(noex)	**---mn**
continue	**---zc**
cxx	**---cxx**
dupsname	**---zd**
endisplaylimit	**---yn**nnn
enexit	**---xt**
enexitdate (*xxx*)	**--xt***xxx*

Long Form	Short Form
enxref (cid)	**--xxx**
enxref (linked)	**--xxe**
enxref (references)	**---xxy**
enxref (sname)	**---xxs**
extname	**---xn**
file (*xxx*)	**---f***xxx*
gmap	**---yg**
inceof	**---zi**
libe	**---b**
lineno	**---l**
list	**---yl**
noenxref	**---!xx**
output *fileid*	**---o***fileid*
pagesize(*nn*)	**---s***nn*
prem	**---p**
print	**---h**
prmap	**---yp**
rtconst	**---r**
smpjclin	**---sj**
smponly	**---szo**
smpxivec	**---sx**
term	**---t**
upper	**---u**
verbose	**---zv**
warn	**---w**
xfnmkeep	**---xf**
xsymkeep	**---xe**

The following is an example of an EXEC statement that invokes COOL with the option NOWARN:

```
// EXEC PGM=COOL#,PARM='!W'
```

COOL Options

This following table 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 7.4 COOL Options

Option	TSO	CMS	OS/390 Batch	USS
---Agather				X
---Ainsert				X
allresident	X	X		X
arlib	X			
auto	X	X	X	
---Bep				X
---Blib				X
cics	X	X		X
cicsvse	X	X		X
clet	X	X	X	X
clet(all)	X	X	X	X
clet(noex)	X	X	X	X
continue	X	X	X	X
cxx	X	X		
dupsname	X	X	X	X
endisplaylimit	X	X	X	X
enexit	X	X	X	X
enexitdata	X	X	X	X
entry	X			
enxref	X	X	X	X
extname	X	X	X	X
files			X	
genmod		X		
global		X		
gmap	X	X	X	X
gos	X	X		
inceof	X	X	X	X
---1				X
---L				X
lib	X			
libe		X		
lineno	X	X	X	X
list	X	X	X	X
lked		X		
lkedname	X		X	

Option	TSO	CMS	OS/390 Batch	USS
load	X			X
loadlib	X			
nocool	X		X	
output		X		
pagesize	X	X	X	X
prem	X	X	X	X
print	X	X		X
prmap	X	X	X	X
rtconst	X	X	X	X
smpjclin	X	X	X	X
smponly	X	X	X	X
smpxivec	X	X	X	X
spe	X	X		X
start		X		
term	X	X	X	
upper	X	X	X	X
verbose	X	X	X	X
warn	X	X	X	X
xfnmkeep	X	X	X	X
xsymkeep	X	X	X	X

-Agather=*prefix* (USS only)
specifies a prefix to be used by COOL in creating a table of symbols whose names begin with the prefix. The effect of this option is the same as if the input contained a GATHER *prefix* control statement. See "The GATHER Statement" on page 163 for more information.

-Ainsert=*symbol* (USS only)
specifies a symbol which must be resolved during COOL processing. If the symbol is not defined in a COOL input file, the symbol will be resolved by autocall. The effect of this option is the same as if the input contained an INSERT *symbol* control statement.

allresident (**-Tallres** under USS)
specifies use of the all-resident library. This option should be specified only when linking an all-resident program. If the application is intended to run under CICS, the **cics** or **cicsvse** option must also be specified so that the correct version of the all-resident library is used.

allowrecool
specifies that the output object deck can be reprocessed by COOL. Therefore, the deck is not marked as already processed by COOL.

The default **noallowrecool** specifies that the output object cannot be reprocessed by COOL. A later attempt to reprocess the deck with COOL will produce an error.

The short form for this option is **-rc**.

Note: COOL does not modify the object deck to enable reprocessing. It is the user's responsibility to determine if a particular object is eligible for reprocessing. △

arlib
in TSO, identifies an AR370 archive containing members that may be included by COOL to resolve unresolved external references. The form of the **arlib** option is as follows:

```
arlib(archive)
```

The *archive* parameter specifies the data set name of the AR370 archive. 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 single quotes.

auto
specifies that COOL should resolve external references by searching the SYSLIB PDS under OS/390 or by searching for TEXT files on an accessible minidisk on CMS. **auto** is the default. **noauto** suppresses resolution of external references from these sources. Note that when an unresolved reference to the symbol *ref* is processed, **auto** attempts to resolve it from SYSLIB(*ref*) on OS/390 or from *ref* TEXT on CMS. The **auto** option is similar to the AUTO option of the CMS load command.

-Bep=*entry* (USS only)
specifies the entry point required for the output load module. The default specification is **-Bep=MAIN**, which assigns the normal C entry point **MAIN**. The name specified must be an external symbol defined in the load module. See "Specifying the Correct Entry Point" on page 167 for more information about entry point specifications. Note that you can suppress the normal entry point of **MAIN** without requesting any other entry-point by specifying **-Bnoep**. You should do this only when one of the object files to be linked includes an ENTRY linkage editor control statement.

-Blib=*library* (USS only)
specifies the name of an OS/390 library to use as a link-edit autocall library. Any number of autocall libraries may be specified, in either object module or load module format. These libraries are not accessed until COOL processing has completed. This means that any references, which can only be resolved from a **-Blib** library, will remain unresolved by COOL. Therefore, the **-Aclet** option should also be specified to allow link processing to occur. Note that the library name should not be preceded by a style prefix such as **//dsn:**, since only OS/390 libraries can be specified with this option.

cics (**-Tcics370** under USS)
specifies that the program should be linked for execution under CICS on OS/390. This option causes appropriate CICS libraries to be added to COOL's list of autocall libraries. The exact libraries that are used depend on whether other options, such as **spe** and **allresident**, were specified.

cicsvse (**-Tcicsvse** under USS)
specifies that the program should be linked for execution under CICS on VSE. This option causes appropriate CICS libraries to be added to COOL's list of autocall libraries. The exact libraries that are used depend on whether other options, such as **spe** and **allresident**, were specified.

clet(all) (**-Aclet** or **-Acletall** under UNIX System Services)
specifies that COOL is to store an output file even if unresolved external reference errors occur. The unresolved references may be for either extended or non-extended names. Neither type of unresolved reference will result in an error

return code. Under UNIX System Services, by default, no output file is generated if any missing symbols or other errors (RC >= 8) are detected by COOL. **clet(noex)** is the default in all other environments that COOL runs under.

In UNIX System Services, the **-Aclet** option is maintained for backward compatibility. It is equivalent to **-Acletall**.

Under OS/390 batch and TSO, the **clet(all)** option takes the following form:

```
clet(all)
```

In CMS, the **clet(all)** option takes the following form:

```
clet  all
```

clet(noex) (**-Acletnoex** under UNIX System Services)
specifies that COOL is to store an output file even if unresolved external reference errors occur. The unresolved references may be for either extended or non-extended names. Unresolved references to non-extended names do not result in an error return code. However, unresolved references to extended names result in an error return code. Under UNIX System Services, by default, no output file is generated if any missing symbols or errors (RC >= 8) are detected by COOL. Otherwise, this option is the default.

Under OS/390 batch and TSO, the **clet(noex)** option takes the following form:

```
clet(noex)
```

In CMS, the **clet(noex)** option takes the following form:

```
clet noex
```

continue (**-Acontinue** under USS)
specifies that processing should continue even if a corrupted AR370 archive is detected.

cxx
specifies that the program being linked includes one or more C++ modules. This option makes the C++ library archive available for resolution of external references. This option should be specified even if the program does not directly use any C++ library functions.

dupsname (**-Adupsname** under USS)
causes COOL to permit the same SNAME to be used in more than one input file. **nodupsname** is the default. Do not specify **dupsname** if any input module uses extended names, or the results will be unpredictable.

enexit (**-Aenexit** under USS)
causes COOL to invoke a user exit without passing any data to the exit. See "User Exit Selection of External Symbols" on page 409 for additional information.

endisplaylimit(nnn)
defines the maximum number of characters used to display extended names in messages and listings. nnn represents the maximum number of characters that can be used to display extended names. nnn can be an asterisk (*) specifying that extended names are to be fully displayed regardless of their length, or it can be a number. The minimum **endisplaylimit** default value of 91 is set internally and cannot be overridden. You can set the default value to a larger number, or use the asterisk to ensure the return of the full-length names, but you cannot set the default value to a number smaller than 91.

OS/390 batch and TSO

endisplaylimit

Under OS/390 batch and TSO, the **endisplaylimit** option takes the following form:

endisplaylimit(nnn)

UNIX System Services

Under UNIX System Services, the **endisplaylimit** option takes the following form:

-Aendisplaylimit=nnn

CMS

Under CMS, the **endisplaylimit** option takes the following form:

endisplaylimit nnn

Cross-Platform

See Chapter 6, "Prelinking C and C++ Programs" in the SAS/C Cross-Platform Compiler and C++ Development System: Usage and Reference for information on the cross-platform implementation of **endisplaylimit**.

enexitdata (**-Aenexitdata=***data* under USS)

causes COOL to invoke a user exit and pass one to eight characters of user-specified data. Under TSO and OS/390 batch, **enexitdata** has the following form:

```
enexitdata(userdata)
```

Under CMS, the **enexitdata** option has the following form:

```
enexitdata userdata
```

See "User Exit Selection of External Symbols" on page 409 for additional information.

entry

In TSO, the keyword

```
entry(name)
```

identifies the program's entry point or enables the linkage editor to determine the entry point. 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 reentrant dynamically loaded module. The actual entry point is #DYNAMN.

entry(dynnr)

for a non-reentrant dynamically loaded module. The actual entry point is #DYNAMNR.

entry(os)

for an OS SPE application with initial function **osmain**. The actual entry point is #OSEP.

entry(oe)

for an SPE application with initial function **oemain**, intended for use under the USS shell or from the **pdscall** utility. The actual entry point is #OEEP.

entry(none)

for allowing 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 (**-Asnamexref**, **-Acidxref**, or **-Alinkidxref** under USS)
When producing object files that contain extended names, COOL produces by default three cross-references that are generated in a table that follows all other COOL output. These three cross-references are **sname**, **cid**, and **linkid**. **sname** is in alphabetical order by the **sname** that uniquely identifies an object file. **cid** displays the extended names in alphabetical order by C identifier. **linkid** displays the extended names in alphabetical order by a link id that COOL assigns. The **enxref** option controls the production of these cross-references so external symbols that are declared but not defined will be printed in the cross-references listing. **noenxref** suppresses the production of all extended names cross-references.

For Release 6.50, when the **references** option is specified for **enxref**, referenced symbols as well as defined symbols are included in the cross-reference listing.

You can use the **references** option (**-Areferences** under UNIX System Services) to modify the behavior of the **sname**, **cid**, or **linkid** cross-reference listing so that cross-references to external symbols that are declared but not defined are included in the listing. You must use the **references** option in conjunction with one or more of these listing options: **sname**, **cid**, or **linkid**. For example:

```
enxref(sname references)
```

specifies an **sname** cross-reference listing that includes cross-references for external symbols that are declared but not defined.

In TSO, the **enxref** option takes the following form:

```
enxref('cross-ref ,cross-ref ,cross-ref ')
```

where cross-ref is **sname**, **cid**, or **linkid**, or the negation. For example:

```
enxref('nosname,cid')
```

suppresses the **sname** cross-reference and enables the CID cross-reference.

Under CMS, the **enxref** option takes the following form:

```
enxref [cross-ref] [cross-ref] [cross-ref]
```

where cross-ref is **sname**, **cid**, or **linkid**, or the negation. For example:

```
enxref nosname cid
```

suppresses the **sname** cross-reference and enables the **cid** cross-reference.

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

```
enxref(cross-ref ,cross-ref ,cross-ref )
```

where cross-ref is **sname**, **cid**, or **linkid**, or the negation. For example:

```
enxref(nosname,cid)
```

suppresses the **sname** cross-reference and enables the **cid** cross-reference.

Under USS, the **sname**, **cid**, and **linkid** cross-references are generated by the **-Asnamexref**, **-Acidxref**, and **-Alinkidxref** options, respectively.

extname (**-Aextname** under USS)
specifies that COOL is to process extended names. **extname** is the default. **noextname** specifies that COOL will not process extended names. For more information on the **extname** option, refer to Chapter 6, “Compiler Options,” on page 101.

files
under OS/390, specifies the first 1 to 3 characters in DDnames referenced by COOL. The **files** option has the following form:

```
files(xxx)
```

where xxx is from 1 to 3 characters. The default is SYS.

genmod
causes the COOL EXEC to create a module file using the specified name and GENMOD options. The **genmod** option takes the following form:

```
genmod [filename [options]]
```

The **genmod** option must follow the use of any other option on the command line. The **genmod** option causes the COOL EXEC to issue the following CMS commands after COOL has created the COOL370 TEXT file:

```
LOAD COOL370 (NOAUTO NOLIBE CLEAR
GENMOD filename   (options
```

where filename is either the filename specified following the **genmod** keyword or the first name specified in the COOL command. If no filenames are specified in the command, the COOL EXEC issues an error message.

global
specifies that the COOL EXEC should query the environment variable TXTLIBS in the GLOBALV group LC370 for the name or names of TXTLIBs that are to be GLOBALed before COOL begins execution. **global** is the default. **noglobal** suppresses automatic query of the environment variable TXTLIBS: the EXEC does not issue a GLOBAL TXTLIB command based on the environment variable. (See "Other Environment Variables" on page 87 for more information.)

gmap (**-Agmap** under USS)
causes a listing of any gathered names in the listing file. The **gmap** option causes the **print** option to be assumed and a listing file to be generated.

gos
specifies that the program should be linked for execution with the GOS (generalized operating system) libraries. This option causes the GOS library to be added to COOL's list of autocall libraries. Note that in TSO, the output of the COOL command is a load module, which may not be suitable for execution under the target operating system.

ignorerecool
specifies that if any marks are detected indicating that COOL has already processed an input object deck, then the marks are to be ignored. If the **ignorerecool** option is specified along with the **verbose** option, then a diagnostic message is issued and processing continues.

The default **noignorerecool** specifies that any mark indicating that COOL has already processed an input object deck should result in an error message and process termination.

The short form for this option is **-ri**.

inceof (**-Ainceof** under USS)
specifies that when an INCLUDE statement is encountered in an included file, the specified file is included, but any data following the statement is ignored. The default is **inceof**. This is compatible with the behavior of the IBM linkage editor. **noinceof** can be specified to allow the use of multiple nested INCLUDE statements.

-l*name* (USS only)
specifies that the archive **lib**name**.a** is to be searched for unresolved external references. **sascc370** looks for the archive in the directories specified by any **-L** option specifications before looking in the **lib** subdirectory of the directory where SAS/C was installed. Note that there must not be a space between **-l** and name. The **-l** option has no effect unless the **-L** option is also specified.

-L*directory* (USS only)
specifies a directory to be searched for archives requested by the **-l** option. The directories referenced by **-L** are searched in the order that the **-L** options appear on the command line. Note that there must not be a space between **-L** and directory.

lib
specifies the data set name of an autocall object library containing functions that are to be linked automatically into the program if referenced. This option takes the following form:

```
lib (dsname)
```

where dsname is the data set name of an autocall object library. 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 single quotes. No final qualifier is assumed for a LIB data set. (Note that load module libraries cannot be used.)

libe
under CMS, the **libe** option causes COOL to search GLOBALed TXTLIBs during automatic symbol resolution. This option is in effect by default and may be turned off by specifying **nolibe**.

lineno (**-Alineno** under USS)
controls line numbering. The COOL **lineno** option is similar to the compiler **lineno** option. You can specify either **lineno** or **nolineno**; the default is **lineno**. If you specify **nolineno** (**-Anolineno** under USS), COOL deletes all the line-number and offset table CSECTs from the output object code.

The line-number and offset table CSECTs are generated by the compiler when the compiler **lineno** option is used. (This is the default for the compiler.) These CSECTs are used by the debugger and run-time library to compute the address of a source line number in a function. If these CSECTs are not present, the debugger cannot break on a source statement and run-time library ABEND tracebacks do not contain function line numbers.

list (**-Alist** under USS)
causes COOL to copy any control statements in its input to the listing file. **list** causes the **print** option to be assumed. The default is **nolist**.

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 the use of all other COOL options 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.

lkedname
specifies the name of the linkage editor to be invoked after COOL has completed. By default, the standard system linkage editor is invoked. This option is provided

to allow sites which run the binder in place of the linkage editor to access the linkage editor instead.

load (**-o** name under USS)
in TSO, names the data set in which the linkage editor stores the output load module. This option takes the form:

```
load (dsname)
```

where *dsname* is the name of the data set in which the linkage editor stores the output load module. 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 single quotes. If the data set name is not specified within three single quotes, it is assumed to be a data set name with a final qualifier of LOAD. Additional considerations follow:

- If the **load** option 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 single quotes, the terminal user is prompted to enter the name of the LOAD data set.

Under USS, the **-o** *name* option specifies the name of a file where the output load module is to be stored. This may be specified as either an HFS file or as a member of an OS/390 PDS. There must be a space between **-o** and *name*. If the output module is stored in an OS/390 PDS, the name should be prefixed with a **//dsn:** or **//tso:** prefix. If no **-o** option is specified, the linked output module is stored in the HFS file **a.out**. Note that if the **sascc370 -c** option is specified, the linkage editor is not invoked, and the **-o** option determines where the compiler's output is stored. See the description of the **object** option in "Option Descriptions" on page 105 for details about the **-c** compiler option.

loadlib (also see **-Blib**)
specifies the data set name of an autocall load library containing modules that are to be linked automatically into the program if referenced. This option takes the following form:

```
loadlib (dsname)
```

where *dsname* is the name of an autocall load library. 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 single quotes. Note that modules in the **loadlib** data set are resolved by the linkage editor and not by COOL. COOL diagnoses any symbols which require resolution from this library 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, since those libraries are stored in load module format.

nocool
specifies that the COOL preprocessor is not to be run. In this case, the input is processed only by the OS/390 linkage editor.

output
specifies the name of the COOL output file. Under CMS, the default COOL output file is COOL370 TEXT A1. If specified, the **output** option must be the last option on the COOL command line and is followed by all or part of a CMS fileid or Shared File System fileid.

A CMS fileid is specified as follows:

```
COOL myprog (OUTPUT filename [[.]filetype [[.]filemode]]
```

where *filename*, *filetype*, and *filemode* identify the COOL output file. For example, the following command writes the COOL output file to AOUT TEXT A1:

```
COOL myprog (output aout text a1
```

If you omit the filetype or filemode, COOL uses TEXT as the default filetype and an asterisk (*) as the default filemode.

An SFS fileid is specified as follows:

```
COOL myprog (output sf:filename [filetype [dirname]]
```

where *dirname* is the complete directory name or the NAMEDEF that has been logically assigned to it. If you omit *filetype* or *dirname*, the default filetype is TEXT and the default directory name is a period (.).

Note: The **output** option may not be used with the **genmod**, **lked**, or **start** option. Each of these options, if specified, must be the last option on the COOL command line. △

pagesize (**-Apagesize=***nn* under USS)
specifies the number of lines to print per page. Under TSO and OS/390 batch, the form of this option is as follows:

```
pagesize(nn)
```

where *nn* is the number of lines. By default, **pagesize(55)** is specified.

Under CMS, the following form is used:

```
pagesize nn
```

prem (**-Aprem** under USS)
specifies that COOL is to remove pseudoregisters from the output object module. Under CMS, the default is **prem**; under OS/390, the default is **noprem**; under USS, the default is **-Anoprem**. Under CMS, the **prem** option allows limitations of the CMS loader to be bypassed. Under OS/390, this option has little use except in certain ILC applications. See the SAS/C Compiler Interlanguage Communication Feature User's Guide for further information.

print (**-Klisting** under USS)
causes COOL to create a listing file that contains a list of the options that are in effect and copies of any diagnostic messages. Under CMS and OS/390, all messages are directed to the listing file and also to **stderr** if the **term** option is specified. Under USS, messages are always sent to **stderr**, as well as the listing file.

Under CMS, use the following syntax:

```
print fileid
```

The fileid must be specified in compressed form, that is, with periods rather than spaces separating the components of the name. If **print** is not specified under CMS, the listing is produced in the file COOL370 COOLLIST.

In TSO, use the following syntax:

```
print(filename)
```

If **print** is specified in TSO without a filename, the listing is directed to the terminal.

Note that the use of other options that write to the listing file cause the **print** option to be assumed, including **list**, **prmap**, **gmap**, and **enxref**. In TSO, if

noprint is specified or defaulted, no listing is generated regardless of other option settings.

Under USS, the **-Klisting** compiler option produces a listing for all phases of the compilation. It also lets you specify the name of the listing file. When you specify any of the following COOL options during compilation, **-Klisting** is assumed: **-Alist**, **-Aprmap**, **-Agmap**, **-Alinkidxref**, **-Asnamexref**, **-Aacidxref**. See Chapter 6, "Compiler Options," on page 101 for details about **-Klisting**.

prmap (**-Aprmap** under USS)
causes COOL to include a pseudoregister map in the listing file. If **prmap** is specified, **print** is implied and will produce a list of options in effect and diagnostic messages, in addition to the output requested by **prmap**. **noprmap** is the default.

rtconst (**-Artconst** under USS)
specifies that COOL is to retain the run-time constants CSECTs in the output object file. This is the default. **nortconst** causes COOL to delete these CSECTs. The resulting object file will be somewhat smaller, but certain information used by the debugger will not be available.

smpjclin (**-Asmpjclin** under USS)
generates a list of linkage editor INCLUDE statements for elements resolved as external references from SMP format libraries. The INCLUDE statements are written to the file identified by the DDname SYSJCLIN. Under CMS, a FILEDEF command can be used to define the DDname, and under the USS shell, the ddn_SYSJCLIN environment variable can be specified.

The list generated by the **smpjclin** option can be used to build an SMPJCLIN file that defines the structure of a software product, including its use of SAS/C Library elements. The SMPJCLIN file is used when an application is to be distributed in System Modification Program (SMP) format. The **smpjclin** option can only be used if you have SMP libraries. For more information, refer to Programmer's Report: SMP Packaging for SAS/C Based Products.

smponly (**-Asmponly** under USS)
causes COOL to build the @EXTVEC# vector described under the **spmxivec** option. The remaining portion of the COOL output is suppressed so that the entire output object file will consist of only the @EXTVEC# CSECT.

smpxivec (**-Asmpxivec** under USS)
causes COOL to build a vector named @EXTVEC# that references sname@. CSECTs that are generated when the **smpxivec** option is specified. This vector table is prepended to the COOL output file. The **smpxivec** option is provided to accommodate SMP update methods. Refer to SAS Programmer's Report: SMP Packaging for SAS/C Based Products for more information on this option.

spe (**-Tspe** under USS)
specifies that the program should be linked for execution with the SPE (Systems Programming Environment) libraries. This option causes the SPE library to be added to COOL's list of autocall libraries.

start
specifies that the COOL EXEC is to issue a LOAD command for COOL370 TEXT, followed by a START command and any of the START options specified. The **start** option must follow the use of all other COOL options on the command line. The format of the **start** option is as follows:

```
start options
```

The **start** option causes the COOL EXEC to issue the following CMS commands after COOL has created the COOL370 TEXT file:

```
LOAD COOL370 (NOAUTO NOLIBE CLEAR
START options
```

where *options* are any START command options specified following the **start** keyword.

term
directs COOL error messages to **stderr** in addition to any other targets. No attempt is made to prevent a message from being sent to the same target via multiple files. In most environments, **stderr** is the user's terminal. For OS/390 batch, **stderr** references the DDname SYSTERM. **noterm** suppresses error message listing to **stderr**. The default is **term**.

upper (**-Aupper** under USS)
produces all output messages in uppercase.

verbose (**-Averbose** under USS)
causes COOL to produce extra messages about its processing, both to the terminal (if **term** is in effect) and to the listing (if a listing is being produced). These messages are useful for determining how symbols are resolved. The default is **noverbose**.

warn (**-Awarn** under USS)
specifies that warning messages (associated with return code 4) are to be issued. **warn** is the default. **nowarn** suppresses warning messages.

xfnmkeep (**-Axfnmkeep** under USS)
specifies that extended function name CSECTs are retained in all input object files. Note that this makes the resulting prelinked object file somewhat larger. By default, **xfnmkeep** is specified.

The extended function name CSECTs may be useful at runtime, if you are using the SAS/C Debugger. If the CSECT containing the extended function name is available, the debugger uses the extended name in displays and accepts the extended name in commands. (Refer to the SAS/C Debugger User's Guide and Reference for more information on the debugger.) Also, if the CSECT that contains the extended name is present, the library abend-handler includes the extended name in abend tracebacks.

See Appendix 7, "Extended Names," on page 405 for additional information about the **xfnmkeep** option.

xsymkeep (**-Axsymkeep** under USS)
specifies that the extended external identifier CSECTs in all input files are retained. Note that this makes the resulting prelinked object file somewhat larger. By default, **noxsymkeep** is specified.

See Appendix 7, "Extended Names," on page 405 for additional information about the **xsymkeep** option.

COOL Control Statements

As mentioned, input to COOL can be either an object data set, control statements, or both. COOL control statements are listed in Table 7.5 on page 162. An explanation of each statement follows the table.

Table 7.5 COOL Control Statements

Control Statement	Explanation
ARLIBRARY*libname*(,*libname* . . .)	*libname* refers to an AR370 archive. Under OS/390, *libname* is a DDname allocated to an AR370 archive. Under CMS, it is the filename of an AR370 archive.
INCLUDE *filename*	*filename* refers to the sequential data set containing the object code to be input to COOL.
INCLUDE*libname*(*member*)	*libname* refers to a partitioned data set. *member* is the member containing the object code to be input to COOL.
INSERT*symbol*	*symbol* is an external symbol. If *symbol* has not been resolved at the end of primary output processing, the automatic call-library mechanism attempts to resolve it. The INSERT statement is also passed to the linkage editor.
GATHER*prefix*	*prefix* is a one-to-six character symbol.

The ARLIBRARY Statement

The ARLIBRARY statement is used to specify the location of an AR370 archive. The following syntax is used:

```
ARLIBRARY name[,name...]
```

The *name* parameters are

- under OS/390, a DDname allocated to an AR370 archive
- under CMS, the filename of an AR370 archive.

COOL adds the libraries specified by the *name* parameters to the list of AR370 archives to be used as autocall input. Refer to “Using AR370 Archives” on page 166 for additional information about using AR370 archives with COOL.

Note: When the ARLIBRARY control statement is used under the USS OS/390 shell, the filename is interpreted as a DDname. You can use an environment variable to supply a pseudo-DDname in this case, as described under "USS I/O Considerations" in Chapter 3 of the SAS/C Library Reference, Volume 1. △

The INCLUDE Statement

The INCLUDE statement specifies the name of one or more additional files for COOL to use as input. The INCLUDE statement has two formats. The first is the following:

```
INCLUDE filename [,...]
```

In TSO and under OS/390 batch, *filename* is a DDname that has been allocated to a sequential data set or member of a PDS. Under CMS, *filename* is the filename of a CMS file. The filetype of the file must be TEXT. The file can be on any ACCESSed disk.

The second format of the INCLUDE statement is the following:

```
INCLUDE libname(member[,member])[,...]
```

In TSO and under OS/390 batch, *libname* is a DDname that has been allocated to a partitioned data set, and *member* is the name of a member of a PDS. Under CMS, *libname* is the name of a TEXT library. The filetype must be TXTLIB. The library can

be on any ACCESSed disk. *member* is the name of a member in the TEXT library. The two formats can be combined on the same statement, for example

```
INCLUDE MAINPROG,MYSUBS(SUB1,SUB2),SYSSUBS(GLBLFNC)
```

An included object file can contain an INCLUDE statement. The specified modules are also included, but any data in the including file after the INCLUDE statement is ignored unless the **noinceof** option is specified.

On CMS, the _INCLUDE environment variable can be used to specify shared file system directories to be searched for included files. For information on the _INCLUDE environment variable, refer to “Specifying Shared File System Directories” on page 86.

Note: When the INCLUDE control statement is used under the USS OS/390 shell, the filename is interpreted as a DDname. You can use an environment variable to supply a pseudo-DDname in this case, as described under "USS I/O Considerations" in Chapter 3 of the SAS/C Library Reference, Volume 1. △

The INSERT Statement

The INSERT statement specifies one or more external symbols that are to be resolved, if necessary, via COOL’s autocall mechanism. The format of the INSERT statement is

```
INSERT symbol [,...]
```

If the symbol specified by INSERT is not resolved after all primary input has been processed, COOL attempts to resolve it by using automatic library call.

The GATHER Statement

The COOL object code preprocessor supports the GATHER control statement. Use of a GATHER control statement in a COOL input file causes COOL to create data tables based on the GATHER statement operands and append these tables to the COOL output object code. The capability of the GATHER statement was designed primarily for the SAS/C++ translator; occasions for using the GATHER statement will be rare.

Statement format

The format of the GATHER control statement is

```
GATHER prefix [,prefix2... ]
```

where prefix is a one-to-six character symbol. The following statements are examples of valid GATHER control statements.

```
GATHER ABC
GATHER INIT,TERM
GATHER I_
```

How COOL processes the GATHER statement

If at least one GATHER control statement is present in a COOL input file, COOL gathers the names of certain External Symbol Dictionary (ESD) items in a list. There is an ESD item for each external defined or referred entry in object code that names an external symbol. For more information, refer to the IBM OS/VS Linkage Editor and Loader, GC26-813. If the name of the ESD item begins with one of the prefixes given by a GATHER control statement, the name is added to a list of names with that prefix. The lists are used to create tables of pointers to the gathered objects that may be referenced in a C program.

Gathered names

COOL inspects ESD items in the input object module(s) (including autocalled modules) that have the following types.

- SD (section definition)
- LD (label definition)
- ER (external reference).

Some ESD items are not considered for gathering. An SD or LD whose name ends with an at sign (@), a colon (:), a dollar sign ($), an equal sign (=), a plus sign (+), a left angle bracket (<), a right angle bracket (>), or a question mark (?) is not considered since the compiler creates data objects with those names. An ER whose name matches a name of a GATHER table (see below) is not considered.

In C, the following objects can create SDs, LDs, and ERs:

LD	label definitions, **`const extern`** objects, and **`extern`** objects when the **`norent`** compiler option is in effect.
ER	references to functions, to **`const extern`** objects, and to **`extern`** objects when the **`norent`** compiler option is in effect.
SD	No C source construct can create an SD that may be gathered. Note that COOL changes any underscore characters (`'_'`) in a prefix to pound signs (`'#'`). This corresponds to the compiler's changing of underscore characters in external names to pound signs.

Listing the gathered names

COOL prints the gathered names in its listing file if the **`gmap`** option has been specified. For a prefix with one or more matching gathered names, COOL prints

```
GATHERED FOR PREFIX "xxxxxx":
   xxxxxxx1
   xxxxxxx2
   xxxxxxx3
```

where **`xxxxxxx1`**, **`xxxxxxx2`**, and so on, are the gathered names. For a prefix for which no matching names were found, COOL prints:

```
GATHERED FOR PREFIX "xxxxxx": (NONE)
```

GATHER tables

For each prefix, COOL creates a GATHER table. A GATHER table is a **`const extern`** structure with the following definition:

```
struct {
   int count;
   void *entry[N]
} xxxxxx$T;
```

The **`count`** field contains the number of gathered objects. If no names were found that matched the prefix, the count field is set to 0. **`entry`** is an array of (4-byte) pointers to the gathered objects. These pointers are in no particular order in the array. **`N`** is the number of gathered objects. **`xxxxxx`** is the prefix that was used to select the objects. For example, if the items in the table are **`__local`** function pointers whose names begin with **`INIT`**, then the GATHER table can be declared as follows:

```
const extern struct {
   int count;
```

```
    __local void (*func[0] )();
} INIT$T;
```

Note that **func** can be declared as an array of length **0**, as shown above. This enables the GATHER table to be declared such that the programmer does not need to know the number of items in the array at compile-time.

CAUTION:

Use the dollars compiler option if your program contains references to GATHER table names. Since the name of the GATHER table always contains the dollar sign ($) character, programs containing references to GATHER table names must be compiled with the **dollars** compiler option. Alternately, the table can be given some other variable name, and the **#pragma map** directive can be used to assign it the external name of **xxxxxx$T**. △

An assembler language view

Each GATHER table is a separate CSECT (SD). The pointer array is a set of 0 or more V-type address constants (ERs).

For example, suppose a program contains declarations for four functions whose names start with **init**: **init0001**, **init0002**, **init0003**, and **init0004** and no declarations or definitions of functions whose names start with **term**. Given the following control statement

```
GATHER INIT,TERM
```

COOL creates GATHER tables as if object code from the following assembler language statements had been included:

```
INIT$T   CSECT
         DC   F'4'
         DC   V(INIT0001)
         DC   V(INIT0002)
         DC   V(INIT0003)
         DC   V(INIT0004)
TERM$T   CSECT
         DC   F'0'
         END
```

Using GATHER tables

Suppose a number of functions need to be invoked upon entry to the main function of a program, and a number of other functions need to be invoked before the main function returns. The programmer specifies that these functions (and only these functions) will have names that begin with the characters **init** or **term**, depending on whether they are to be invoked upon program startup or termination, respectively. The number of functions will be unknown at compile time, as will the complete names. If all of the functions are located in the primary load module, they can be called via **__local** function pointers. The following GATHER control statement causes COOL to produce GATHER tables for these two sets of functions:

```
GATHER INIT,TERM
```

The following program fragment illustrates how the main function might call the startup and termination functions via the GATHER tables.

```
const extern struct {
                int count;
                __local void (*func[0] )();
```

```
                }  init$t, term$t;

int main()
{
   int i, rc;

   for (i = 0; i < init$t.count; i++) (*init$t.func[i] )();
   .
   .
   .
   for (i = 0; i < term$t.count; i++) (*term$t.func[i] )();
   return rc;

}
```

For more information about **__local** function pointers, refer to "Local Function Pointers" on page 51.

Using AR370 Archives

An AR370 archive is a collection of object modules that can be used by COOL to resolve external references, including external references to extended names. AR370 archives are created and maintained by the AR370 archive utility.

CAUTION:

Use ar370 to access AR370 archives. Do not attempt to create or modify an AR370 archive using any tools other than AR370 and UPDTE2AR. AR370 archives are stored in a binary format and will be rendered unusable if modified by a program unfamiliar with this structure, such as a text editor. △

COOL enables you to use AR370 archives as follows:

- Under OS/390, the ARLIBRARY control statement specifies a DDname associated with an AR370 archive. However, instead of using an ARLIBRARY statement, you can allocate one or more AR370 archives to the DDname SYSARLIB. In TSO, you should use the **arlib** option of the CLK370 CLIST to specify additional AR370 archives rather than the DDname SASARLIB.
- Under CMS, the ARLIBRARY control statement specifies the filename of an AR370 archive. However, instead of using an ARLIBRARY statement, you can specify the filename of an AR370 archive on the COOL command line.
- Under the USS Shell, you can specify the filename of one or more AR370 archives on the **sascc370** command line. It is also possible to use the ARLIBRARY control statement to specify an archive to be used under the shell. See "The ARLIBRARY Statement" on page 162 for more information.

Using the ARLIBRARY Control Statement

The ARLIBRARY control statement is used to add an archive to the list of AR370 archives to be used as autocall input. For example, the following statement adds ALPHA to the list of archives:

```
ARLIBRARY ALPHA
```

ALPHA could be either an OS/390 DDname or a CMS filename. Refer to "The ARLIBRARY Statement" on page 162 for additional information.

Specifying Archives from the Command Line

Under CMS, you can specify AR370 archives from the COOL command line. The COOL EXEC invokes the COOL object code preprocessor and can optionally invoke the CMS LOAD, START, GENMOD, or LKED command. The format is as follows:

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

where *filename1*, *filename2*, and so on are the names of the files that are to be the primary input to COOL. Each file should have a filetype of TEXT and contain either object code or COOL/linkage editor control statements.

A *filename* argument can also be the name of an AR370 archive with a filetype of A. For example, the following command specifies two AR370 archives named LIB1 and LIB2:

```
COOL [filename1 [filename2 ...]] LIB1 LIB2 [(options[)]]
```

COOL adds the specified AR370 archives to the list of AR370 archives to be used as autocall input.

When a filename appears on the command line, COOL checks to see if the filename identifies an AR370 archive before checking for a TEXT file. Thus, in this example, if both LIB1 A and LIB2 TEXT are found, LIB1 A will be processed and LIB2 TEXT ignored.

Similarly, under the USS shell, you can specify the names of AR370 archives to be processed by COOL on the **sascc370** command line. For example, the following command requests prelinking and linking of the object module **main.o**, resolving references from the archive named **subs.a**:

```
sascc370 -o shgame main.o subs.a
```

Specifying the Correct Entry Point

Because of the variety of ways a SAS/C program can be constructed, the same entry point is not appropriate for all programs. This section describes how to determine what entry point your application requires and then describes how to specify this entry point when you link the program.

- In the case of a normal, single load module SAS/C application with no special requirements, the program entry point is the symbol **MAIN**. In most cases, you need not take any special action to have this entry point selected.
- The module you are linking may be a dynamically loadable component of a multi-load-module program. In this case, the entry point name depends on whether you compiled with the **norent** option or with one of **rent** or **rentext**. In the **norent** case, the entry point is #DYNAMNR. In the **rent** or **rentext** case, the entry point is #DYNAMN. Under TSO, OS/390 batch and the shell, you can often specify a keyword to ensure that the correct entry point is selected.
- You may have written your program to be called from assembler language, as described in Chapter 11, “Communication with Assembler Programs,” on page 209. Some of these calls require one of the entry points $MAINC or $MAINO, as described in this chapter. In this case, you must explicitly specify the correct entry point.
- You may have compiled your initial function with the **indep** option. If so, this function must be specified as the entry point. Since the name of this function is not fixed, you must explicitly specify the correct entry point.

- You may be linking your application with the SPE library (see Chapter 14, "Systems Programming with the SAS/C Compiler," on page 273) and using the standard front-end for your environment. In this case, the correct entry point will be either #OSEP, #CMSEP, #CICSEP or #OEEP, depending on whether the program is intended for execution under OS/390, CMS, CICS, or the USS shell. Under TSO, OS/390 batch and the shell, you can often specify a keyword to ensure that the correct entry point is used.
- You may be linking your application with the SPE library and using a custom front-end. In this case, you must specify your own front-end as the entry point. Since this name cannot be predicted, you must explicitly specify the entry point.

The correct entry point can always be specified via a linkage editor control card or the RESET option of the CMS LOAD command. Additionally, when you use most of the SAS/C batch cataloged PROCs or the COOL TSO CLIST, you can often use the **entry** keyword to specify the entry point; when you compile under the USS shell, you can use the **-Bep** option of **sascc370** for this. To illustrate:

OS/390 batch

```
// EXEC LC370L,PARM.LKED='LIST,MAP',ENTRY=MAIN
```

TSO

```
cool example(init) list map rent entry(dyn)
```

USS

```
sascc370 -Bep='#OEEP' -Tspe example/init.c
```

The **entry** keyword may have any of the following values:

MAIN	The module is an ordinary C main program.
DYN	The module is a **rent**- or **rentext**-compiled dynamically loadable module.
DYNNR	The module is a **norent**-compiled dynamically loadable module
OS	The module is an SPE application using the default OS/390 start-up code
OE	The module is an USS SPE application using the default USS start-up code.
CSPE	The module is a CICS SPE application using the default CICS start-up code. This option is available only with the CICS batch PROCs (for example, LCCCPCL).
NONE	The entry point to the module is specified by an input ENTRY control statement provided by the user.

If no **entry** option is specified, the default is always MAIN.

Note: The **-Bep** option of the **sascc370** shell command, unlike the **entry** keyword for batch and TSO, requires that the actual entry point be specified, not a code. Thus, the example above specifies **-Bep='#OEEP'**, not **-Bep=OE**. △

SAS/C Library Names

Table 7.6 shows the SAS/C Libraries that can be linked with your program, for both OS/390 and CMS. This information may be valuable if you are linking your program without using a standard batch PROC or the COOL CLIST or EXEC. Note that in general, you should define the base run-time library and one other library, depending

on the intended execution environment, as autocall libraries. Optionally, a version of the all-resident library can be added. Do not attempt to link using more than one environmental library at once, as the effects are unpredictable. For instance, never specify both the standard library and the SPE library together as autocall libraries.

Note that on OS/390 the name shown below must be augmented by a site-assigned prefix. For instance, the name of the base library is shown as *prefix*.BASEOBJ. The prefix part of the filename will be replaced by a prefix chosen by your site. For instance, if your site uses the prefix VDR.SASC600, then the full name of the base library will be VDR.SASC600.BASEOBJ.

For OS/390, two library names are shown for some libraries. The names ending in OBJ are object module libraries. The names ending in LIB or SUB are the equivalent load module libraries.

Table 7.6 SAS/C Libraries

Library Contents	OS/390 Name	CMS Name
Base run-time library (modules valid in all systems)	*prefix*.BASEOBJ *prefix*.BASELIB	LC370BAS TXTLIB
Standard run-time library (normal system environment)	*prefix*.STDOBJ *prefix*.STDLIB	LC370STD TXTLIB
Standard library (GOS environment)	*prefix*.GOSOBJ	LC370GOS TXTLIB
SPE run-time library (SPE environment)	*prefix*.SPEOBJ *prefix*.SPELIB	LC370SPE TXTLIB
CICS run-time library (OS/390 environment)	*prefix*.CICSOBJ *prefix*.CICSLIB	LC370CIC TXTLIB
CICS run-time library (VSE environment)	*prefix*.VSEOBJ	LC370VSE TXTLIB
CICS SPE library	*prefix*.CICS.SPEOBJ *prefix*.CICS.SPELIB	LC370SPC TXTLIB
All—resident library (non-CICS)	*prefix*.ARESOBJ	LCARES TXTLIB
All—resident library (CICS)	*prefix*.CICS.ARESOBJ	CICSARES TXTLIB
ILC library	*prefix*.ILCOBJ *prefix*.ILCSUB	see notes

Note: The ILC library should not be used as an autocall library. Under CMS these files are organized as a collection of TEXT files rather than as a TXTLIB. See the SAS/C Compiler Interlanguage Communication Feature User's Guide for more information on linking ILC programs. △

CHAPTER

8

Executing C Programs

Introduction

This chapter contains instructions for running your SAS/C program in TSO, from the UNIX System Services (USS) shell, under CMS, and under OS/390 batch.

Executing C Programs in TSO

Any C program can be run in TSO by use of the standard TSO CALL command. Note, however, that when a C program is executed using CALL, certain TSO oriented features are not available. In particular, the command name is not known to the program.

Alternately, your installation may provide a higher level of support for calling C programs from TSO. There are two higher levels available:

- calling via the TSO C command
- calling as a standard TSO command.

The following are examples of how you call a sample program from TSO at each level of support. In the examples, the library option **=w** overrides the program's use of the **quiet** function to suppress warnings. **<input** redirects **stdin** from the DDname INPUT. The final parameter, **-z**, is an argument to the program. See Chapter 9, "Run-Time Argument Processing," on page 185 for information about program parameters.

□ Call your C program via the TSO CALL command as follows:

```
call library.name(tsoexam) '=w <input -z'
```

This statement uses the TSO CALL command to execute the program **tsoexam**. The data set containing **tsoexam** is **library.name**. The program parameters are surrounded by single quotes.

Note: The CALL command translates program arguments to uppercase unless you specify the ASIS keyword. △

Both forms of the optional support (illustrated in the next two examples) require that your program data set be allocated to the DDname CPLIB. The examples assume that this has been done before the programs are invoked.

□ Call your C program via the C command as follows:

```
c tsoexam =w <input -z
```

The C command calls **tsoexam**, passing the parameters that follow the program name to the program.

□ Call your C program as a standard TSO command as follows:

```
tsoexam =w <input -z
```

The **tsoexam** program is invoked as if it were a standard TSO command. Again, the parameters that follow the program name are passed to the program.

When you execute any C program in TSO, the transient run-time library must be allocated to the DDname CTRANS or installed in the system link list. Your installation will probably cause it to be allocated automatically; if not, you should use the TSO ALLOCATE command to associate this library with the DDname CTRANS.

Using the Debugger

Before using the SAS/C Debugger, it is recommended that you compile with the **debug** option. Debugger access to program source and variables is permitted only if the C program was compiled with **debug**. Note that if you compile with **dbhook** rather than with **debug**, debugging is limited to commands that do not involve source or SAS/C Debugger User's Guide and Reference variable access. If you compile without specifying either **debug** or **dbhook**, use of the debugger is limited to tracing or stopping execution at subroutine call and return.

In TSO, the debugger expects the debugger symbol table file to be allocated to the DDname DBGLIB at the start of execution. Alternately, you can use the debugger **set search** command in the debugger profile to inform the debugger of the location of symbol tables.

Refer to the SAS/C Debugger User's Guide and Reference for information on running the debugger in TSO.

Executing C Programs from the USS Shell

To run a SAS/C program under the USS shell, the program must reside in an executable HFS file, and its directory must be included in the value of the PATH environment variable. If these conditions are satisfied, you can execute the program simply by typing the name and any options at the shell prompt. If the program's directory is not in your PATH, you can still run it, but you must give the full pathname. You can also call a program in a PDS from the shell by using the pdscall utility. See "Using pdscall" on page 173 for details.

The following is an example of calling a program from the shell.

```
shlexam =w -z <input
```

In the example, the library option **=w** overrides the program's use of the **quiet** function to suppress warnings, and **-z** is a program option. The redirection, **<input**, is processed by the shell, not by the SAS/C Library, and causes the standard input file to be redirected from the file named INPUT in the current directory.

Note: Because redirections are interpreted by the shell rather than by the library, you cannot redirect files to nonstandard filenames, such as ones whose names start with **//**. △

Before you can run any SAS/C program under the shell, the SAS/C Transient Library must be available. Your site may make the library available automatically, by installing it into the system link list, or by using **/etc/profile** to make it available when you use the **omvs** command to startup the shell. If your site does not make the SAS/C Library available, then you must do one of the following prior to invoking a SAS/C program:

- Define the environment variable ddn_CTRANS as the data set name of the SAS/C Transient Library.
- Define the environment variable STEPLIB as a list of data set names, one of which is the SAS/C Transient Library.

For example, if your site installed the transient library as VDR.SASC.LINKLIB, then you could specify either of the following:

```
export ddn_CTRANS=vdr.sasc.linklib
export STEPLIB='vdr.sasc.linklib:sys1.favorite.linklib'
```

The latter command defines SYS1.FAVORITE.LINKLIB as a STEPLIB in addition to the SAS/C Library.

Note that using ddn_CTRANS may improve performance, particularly if you frequently invoke the standard utilities, which are not written using SAS/C and therefore do not require transient library access.

Using pdscall

pdscall is an USS shell command that can be used to run SAS/C programs stored in a PDS. The programs are called using USS **exec**-linkage, so that their behavior should be the same as if the program were copied to the hierarchical file system and invoked as a shell command.

The syntax of **pdscall** is as follows:

```
pdscall pgmname options
```

The *pgmname* should be a fully qualified OS/390 data set name followed by a member name in parentheses. If no member name is specified, the member TEMPNAME is assumed. The *pgmname* may be specified in either uppercase or

lowercase, and it should be enclosed in quotes to prevent shell interpretation of the parentheses around the member name. Following the program name may optionally appear one or more program options. These are passed to the program unmodified.

The exit status of **pdscall** is the same as that of the invoked program.

Note that **pdscall** may produce successful results with programs that are not written in SAS/C, but that correct results are not guaranteed.

Using the Debugger

Before using the SAS/C Debugger, it is recommended that you compile with the **debug** option. Debugger access to program source and variables is permitted only if the program was compiled with **debug**. Note that if you compile with **dbhook** rather than with **debug**, debugging is limited to commands that do not involve source or variable access. If you compile without specifying either **debug** or **dbhook**, use of the debugger is limited to tracing or stopping execution at function call and return.

You should use the **set search** command in your profile to inform the debugger of the locations of debugger symbol tables. See SAS/C Software: Changes and Enhancements, Release 7.00 for information on **set search**.

Note that use of the **sascdbg** command is required to debug a program running under the shell. The run-time option **=debug** will be ignored if specified for a program running under the shell.

Executing C Programs under CMS

There are many ways to invoke a C program under CMS. The most frequently used method is to create a MODULE file with the GENMOD command and then invoke the program as any other CMS command. The following example shows how a MODULE file named CMSEXAM can be invoked:

```
CMSEXAM =w <input.file -z
```

In the example, the library option **=w** suppresses the **quiet** function. **<input.file** redirects **stdin** from the file INPUT FILE. **--z** is a program argument that is to be passed to the **main** function via the **argv** vector. See Chapter 9, “Run-Time Argument Processing,” on page 185 for information about the types of program parameters.

C programs in TEXT files can be invoked with the LOAD and START commands, as follows:

```
LOAD CMSEXAM
START * =w <input.file -z
```

The program parameters follow the asterisk (*) in the START command.

When you execute any C program under CMS, the transient run-time library must be on an accessed disk or in a segment available to your virtual machine. Your installation probably makes it available automatically; if not, ask your SAS Software Representative for C compiler products about how to get access to the transient library.

Using the Debugger

Before using the SAS/C Debugger, it is recommended that you compile with the **debug** option. Debugger access to program source and variables is permitted only if the C program was compiled with **debug**. Note that if you compile with **dbhook** rather than with **debug**, debugging is limited to commands that do not involve source or variable

access. If you compile without specifying either **debug** or **dbhook**, use of the debugger is limited to tracing or stopping execution at subroutine call and return. For information on running the debugger under CMS, refer to the SAS/C Debugger User's Guide and Reference.

CMS Parameter Lists

As with any C program, the program parameters are transferred to the **main** function via the **argv** vector. C programs under CMS generally use the untokenized parameter list to create the **argv** vector. (See Chapter 9, "Run-Time Argument Processing," on page 185 for more information on the **argv** vector.) The untokenized parameter list does not alter the program parameters, in case or in length. For example:

```
cmsexam three very-long parameters
```

In this invocation of a C program, the **main** function receives pointers to these strings:

```
three
very-long
parameters
```

In some cases, however, CMS provides only a tokenized parameter list. If this occurs, the C program parameters are converted to uppercase, and each token is truncated to eight characters. Given the command line above, provided as a tokenized parameter list, the **main** function receives pointers to the following strings:

```
THREE
VERY-LON
PARAMETE
```

Programs that can be invoked by CMS in such a way should be prepared to accept tokenized parameters. Note that C programs invoked via the CMS EXEC processor (as opposed to the EXEC2 processor or REXX) receive tokenized parameters.

Standard file redirections

Under CMS, the standard files **stdin** and **stdout** can be redirected to nonterminal files. (See Chapter 9, "Run-Time Argument Processing," on page 185 for more information.) A typical redirection of **stdin** from a disk file might look like the following:

```
cmsexam <data.file.b
```

If only the tokenized parameter list is available, the redirection parameter is truncated to <DATA.FI, which probably causes an error to occur when **stdin** is opened. Therefore, the library accepts the following alternate redirection form:

```
cmsexam <(data file b)
```

The fileid does not use periods and is entirely enclosed by parentheses. Using this form of redirection parameter does not cause truncation of the fileid in a tokenized parameter list.

Executing C Programs under OS/390 Batch

The following sections discuss the cataloged procedures provided to execute C programs immediately after the link-edit step.

Using Cataloged Procedures to Execute C Programs

You can use one of the cataloged procedures LC370CLG or LC370LG to execute a C program immediately after it is link-edited. Neither of these procedures runs COOL. If you need to run COOL before link-editing and executing a program, because it is reentrant or uses extended names, use the LC370LRG cataloged procedure.

Example Code 8.1 on page 176 shows how to link-edit and execute a C program. Example Code 8.2 on page 176 and Example Code 8.3 on page 177 show the expanded JCL for the cataloged procedures that execute non-reentrant C programs LC370LG and LC370CLG, respectively.

Example Code 8.1 Sample JCL for Linking and Executing a C Program Using the Cataloged Procedure LC370LG

```
//JOBNAME  JOB   job card information
//*-------------------------------------------------------------
//*   LINK AND RUN A C PROGRAM
//*-------------------------------------------------------------
//STEP1        EXEC LC370LG
//LKED.SYSLMOD DD DISP=SHR,DSN=your.load.library(member)
//LKED.SYSIN   DD DISP=SHR,DSN=your.object.library(member)
/*
```

The LC370LG Cataloged Procedure

Expanded JCL for the LC370LG procedure is illustrated in Example Code 8.2 on page 176. This JCL is correct as of the publication of this guide. However, it may be subject to change.

Example Code 8.2 Expanded JCL for the LC370LG Procedure

```
//LC370LG  PROC ENTRY=MAIN,ENV=STD,
//         CALLLIB='SASC.BASELIB',
//         SYSLIB='SASC.BASELIB'
//*************************************************************
//*  PRODUCT:  SAS/C                                          ***
//*  PROCEDURE:  LINKAGE AND EXECUTION                        ***
//*  DOCUMENTATION:  SAS/C COMPILER AND LIBRARY USER'S GUIDE   ***
//*  FROM:  SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC     ***
//*************************************************************
//*
//* ******************************************************************

//* ENV=STD:     MODULE RUNS IN THE NORMAL C ENVIRONMENT
//* ENV=SPE:     MODULE USES THE SYSTEMS PROGRAMMING ENVIRONMENT
//* ENTRY=MAIN:  MODULE IS A NORMAL C MAIN PROGRAM
//* ENTRY=OS:    MODULE IS AN OS SPE APPLICATION
//* ENTRY=NONE:  ENTRY POINT TO BE ASSIGNED BY USER
//* ******************************************************************
//LKED     EXEC PGM=LINKEDIT,PARM='LIST,MAP',REGION=1536K
//SYSPRINT DD SYSOUT=A,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSTERM  DD SYSOUT=A
//SYSLIN   DD DSN=SASC.BASEOBJ(EP@&ENTRY),
```

```
//            DISP=SHR
//         DD DDNAME=SYSIN
//SYSLIB   DD DSN=SASC.&ENV.LIB,
//            DISP=SHR             STDLIB OR SPELIB
//         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))
//GO       EXEC PGM=*.LKED.SYSLMOD,COND=(4,LT,LKED)
//STEPLIB  DD DSN=SASC.LINKLIB,
//            DISP=SHR C TRANSIENT LIBRARY
//SYSPRINT DD SYSOUT=A
//SYSTERM  DD SYSOUT=A

//DBGLOG   DD SYSOUT=A
//SYSTMPDB DD UNIT=SYSDA,SPACE=(TRK,25)  VS1 ONLY
```

Note: The symbolic parameter SYSLIB refers to the autocall library at your installation. △

The LC370CLG Cataloged Procedure

Expanded JCL for the LC370CLG procedure is illustrated in Example Code 8.3 on page 177. This JCL is correct as of the publication of this guide. However, it may be subject to change.

Example Code 8.3 Expanded JCL for the LC370CLG Procedure

```
//LC370CLG PROC ENTRY=MAIN,ENV=STD,
//         CALLLIB='SASC.BASELIB',
//          MACLIB='SASC.MACLIBH',
//          SYSLIB='SASC.BASELIB'
//*************************************************************
//*  PRODUCT:  SAS/C                                          ***
//*  PROCEDURE:  COMPILATION, LINKAGE, AND EXECUTION          ***
//*  DOCUMENTATION:  SAS/C COMPILER AND LIBRARY USER'S GUIDE   ***
//*  FROM:  SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC     ***
//*************************************************************
//*
//* ***************************************************************

//* ENV=STD:     MODULE RUNS IN THE NORMAL C ENVIRONMENT
//* ENV=SPE:     MODULE USES THE SYSTEMS PROGRAMMING ENVIRONMENT
//* ENTRY=MAIN:  MODULE IS A NORMAL C MAIN PROGRAM
//* ENTRY=OS:    MODULE IS AN OS SPE APPLICATION
//* ENTRY=NONE:  ENTRY POINT TO BE ASSIGNED BY USER
//* ***************************************************************
//C        EXEC PGM=LC370B
//STEPLIB  DD DSN=SASC.LOAD,
//            DISP=SHR        COMPILER LIBRARY
//         DD DSN=SASC.LINKLIB,
```

```
//            DISP=SHR       RUNTIME LIBRARY
//SYSTERM  DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
//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=&MACLIB,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
//LKED     EXEC PGM=LINKEDIT,PARM='LIST,MAP',COND=(8,LT,C)
//SYSPRINT DD SYSOUT=A,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSTERM  DD SYSOUT=A
//SYSLIN   DD DSN=*.C.SYSLIN,DISP=(OLD,PASS),VOL=REF=*.C.SYSLIN
//         DD DSN=SASC.BASEOBJ(EP@&ENTRY),
//            DISP=SHR
//         DD DDNAME=SYSIN

//SYSLIB   DD DSN=SASC.&ENV.LIB,
//            DISP=SHR                  STDLIB OR SPELIB
//         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))
//GO       EXEC PGM=*.LKED.SYSLMOD,COND=((8,LT,C),(4,LT,LKED))
//STEPLIB  DD DSN=SASC.LINKLIB,
//            DISP=SHR     C TRANSIENT LIBRARY
//SYSPRINT DD SYSOUT=A
//SYSTERM  DD SYSOUT=A
//DBGLOG   DD SYSOUT=A
//DBGLIB   DD DSN=*.C.SYSDBLIB,DISP=(OLD,PASS),VOL=REF=*.C.SYSDBLIB
//SYSTMPDB DD UNIT=SYSDA.,SPACE=(TRK,25)     VS1 ONLY
```

Note the following about this example:

- The symbolic parameter MACLIB refers to the data set name chosen by your installation for the macro library. The symbolic parameter SYSLIB refers to the autocall library at your installation. Do not override these parameters.
- When you use LC370CLG more than once in a job, provide overriding JCL (DISP=(OLD,PASS)) to reuse the compiler SYSLIN data set (&&OBJECT) in all but the first instance.
- When you override SYSPRINT to reference a disk data set, the data set disposition must be MOD. The data set cannot be a member of a PDS.
- ENV=STD is the default and specifies the standard OS/390 environment. ENV=SPE should be used for an SPE application. (See "Selecting the program environment" on page 139 for more information.)
- See "Linking Programs under OS/390 Batch" on page 138 for information on using the ENTRY symbolic parameter. Note that the only valid specifications for LC370CLG are ENTRY=MAIN, ENTRY=OS and ENTRY=NONE, since only these can produce a separately executable OS/390 load module.

The LC370LRG Cataloged Procedure

The LC370LRG cataloged procedure is similar to LC370LR, with the addition of a GO step. Expanded JCL for the LC370LRG procedure is illustrated in Example Code 8.4 on page 179. This JCL is correct as of the publication of this guide. However, it may be subject to change.

Example Code 8.4 Expanded JCL for the LC370LRG Procedure

```
//LC370LRG PROC ENV=STD,ALLRES=NO,
//         CALLLIB='SASC.BASEOBJ',
//         SYSLIB='SASC.BASEOBJ'
//*************************************************************
//*  PRODUCT:  SAS/C                                          ***
//*  PROCEDURE:  COOL LINKAGE EDITOR PREPROCESSOR, LINK EDIT  ***
//*              AND EXECUTE                                  ***
//*  DOCUMENTATION:  SAS/C COMPILER AND LIBRARY USER'S GUIDE  ***
//*  FROM:  SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC    ***
//*************************************************************
//* ENV=STD:     MODULE RUNS IN THE NORMAL C ENVIRONMENT
//* ENV=SPE:     MODULE USES THE SYSTEMS PROGRAMMING ENVIRONMENT
//* ****************************************************************
//LKED     EXEC PGM=COOLB,PARM='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=A,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSTERM  DD SYSOUT=A
//SYSLIN   DD UNIT=SYSDA,DSN=&&LKEDIN,SPACE=(3200,(20,20)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSLIB   DD DDNAME=AR#&ALLRES       ARESOBJ OR STDOBJ OR SPEOBJ
//         DD DSN=SASC.&ENV.OBJ,
//            DISP=SHR                STDOBJ OR SPEOBJ
//         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.ARESOBJ,
//            DISP=SHR
//GO       EXEC PGM=*.LKED.SYSLMOD,COND=(4,LT,LKED)
//STEPLIB  DD DSN=SASC.LINKLIB,
//            DISP=SHR C TRANSIENT LIBRARY
//SYSPRINT DD SYSOUT=A
//SYSTERM  DD SYSOUT=A
//DBGLOG   DD SYSOUT=A
//SYSTMPDB DD UNIT=SYSDA,SPACE=(TRK,25)  VS1 ONLY
```

Note the following about this example:

□ The symbolic parameter SYSLIB refers to the autocall library at your installation.

- ENV=STD is the default and specifies the standard OS/390 environment. ENV=SPE should be used for an SPE application. (See the section "Selecting the program environment" on page 139 for more information.)
- A SYSLDLIB DD statement may be provided to define one or more autocall libraries in load module format. Any references to members of SYSLDLIB are left unresolved by COOL and are resolved by the linkage editor.

The LC370CRG Cataloged Procedure

The LC370CRG cataloged procedure is similar to LC370CLR, with the addition of a GO step. Expanded JCL for the LC370CRG procedure is illustrated in Example Code 8.5 on page 180. This JCL is correct as of publication of this guide. However, it may be subject to change.

Example Code 8.5 Expanded JCL for the LC370CRG Procedure

```
//LC370CRG  PROC ENV=STD,ALLRES=NO,
//          CALLLIB='SASC.BASEOBJ',
//           MACLIB='SASC.MACLIBC',
//           SYSLIB='SASC.BASEOBJ'

//*************************************************************
//*  NAME:  LC370CRG                          (LC370CRG)     ***
//*  PRODUCT:  SAS/C                                         ***
//*  PROCEDURE:  COMPILATION, PRE-LINK, LINKAGE, AND EXECUTION ***
//*  DOCUMENTATION:  SAS/C COMPILER AND LIBRARY USER'S GUIDE   ***
//*  FROM:  SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC     ***
//*************************************************************
//*
//* ******************************************************************
//* ENV=STD:     MODULE RUNS IN THE NORMAL C ENVIRONMENT
//* ENV=SPE:     MODULE USES THE SYSTEMS PROGRAMMING ENVIRONMENT
//* ******************************************************************
//C         EXEC PGM=LC370B
//STEPLIB   DD DSN=SASC.LOAD,
//             DISP=SHR        COMPILER LIBRARY
//          DD DSN=SASC.LINKLIB,
//             DISP=SHR       RUNTIME 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,DCB=(RECFM=FB,LRECL=80)
//SYSLIB    DD DSN=&MACLIB,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
//LKED      EXEC PGM=COOLB,PARM='LIST,MAP',COND=(8,LT,C),REGION=1536K
//STEPLIB   DD DSN=SASC.LOAD,
//             DISP=SHR        COMPILER LIBRARY
//          DD DSN=SASC.LINKLIB,
```

```
//            DISP=SHR       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)
//SYSLIB   DD DDNAME=AR#&ALLRES        ARESOBJ OR STDOBJ OR SPEOBJ
//         DD DSN=SASC.&ENV.OBJ,
//            DISP=SHR                 STDOBJ OR SPEOBJ
//         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))
//SYSIN    DD DSN=*.C.SYSLIN,DISP=(OLD,PASS),VOL=REF=*.C.SYSLIN
//AR#NO    DD DSN=SASC.&ENV.OBJ,
//            DISP=SHR
//AR#YES   DD DSN=SASC.ARESOBJ,
//            DISP=SHR
//GO       EXEC PGM=*.LKED.SYSLMOD,COND=((8,LT,C),(4,LT,LKED))
//STEPLIB  DD DSN=SASC.LINKLIB,
//            DISP=SHR     C TRANSIENT LIBRARY
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//DBGTERM  DD SYSOUT=*
//DBGLOG   DD SYSOUT=*
//DBGLIB   DD DSN=*.C.SYSDBLIB,DISP=(OLD,PASS),VOL=REF=*.C.SYSDBLIB
//SYSTMPDB DD UNIT=SYSDA,SPACE=(TRK,25)      VS1 ONLY
```

Note the following about this example:

- The symbolic parameter SYSLIB refers to the autocall library at your installation.
- ENV=STD is the default and specifies the standard OS/390 environment. ENV=SPE should be used for an SPE application. (See the section “Selecting the program environment” on page 139 for more information.)
- A SYSLDLIB DD statement may be provided to define one or more autocall libraries in load module format. Any references to members of SYSLDLIB are left unresolved by COOL and are resolved by the linkage editor.

Run-Time JCL Requirements

To run a C program, some or all of the DD statements summarized in Table 8.1 on page 181 may be required.

Table 8.1 Data Definition Statements for Program Execution under OS/390

DDname	Purpose
STEPLIB or JOBLIB	Must include the run-time library in the concatenation unless it is already in your system libraries or defined by CTRANS.
CTRANS	May be used to define the run-time library if not defined in JOBLIB or STEPLIB or in your system libraries.

DDname	Purpose
SYSTERM	**stderr** (standard error) output. Highly recommended since run-time library error messages go to **stderr**.
SYSPRINT	**stdout**. Required only if the program writes to**stdout**.
SYSIN	**stdin**Required only if the program reads from**stdin**.
DGBLIB	Debugger symbol table file.
DBGSRC	Source library data set or data set concatenation.
DBGLOG	A log of messages from the debugger.
DBGIN	A file containing commands to be read by the debugger.

CTRANS may be used to define the run-time library if not defined in JOBLIB or STEPLIB or in your system libraries.

The PARM keyword of the EXEC statement can be used to specify program arguments, which are passed by the **argc/argv** interface to the main program. The PARM string can also contain library arguments, standard file redirections, and environment variables.

A SYSUDUMP card should be included if a dump is desired.

The DBGLIB, DBGSRC, DBGLOG, and DBGIN DDnames are explained in detail in the SAS/C Debugger User's Guide and Reference.

Example Code 8.6 on page 182 illustrates how to execute a C program that writes to standard output.

Example Code 8.6 Sample JCL for Program Execution

```
//JOBNAME  JOB  job card information
//*--------------------------------------------------------------
//*   RUN A C PROGRAM
//*--------------------------------------------------------------
//JOBLIB   DD DISP=SHR,DSN=your.load.library
//         DD DISP=SHR,DSN=runtime.transient.library
//STEP1    EXEC PGM=MVSEXAM,PARM='=W <INPUT -z'
//SYSPRINT DD SYSOUT=class
//SYSTERM  DD SYSOUT=class
//INPUT    DD DSN=your.input.data,DISP=SHR
/*
```

Using the GETENV and PUTENV TSO Commands

The TSO commands PUTENV and GETENV are provided to enable you to access or set SAS/C EXTERNAL environment variables. (Also see Chapter 4, "Environment Variables," in SAS/C Library Reference, Volume 1.) Your site has probably installed these commands into the system link list, in which case you can use them as you use all other TSO commands. If your site has not installed these commands, you will get a **COMMAND NOT FOUND** message when you attempt to use them. In this case, you can still access these commands with the TSO CALL command.

The GETENV Command

The GETENV command is used to print the values of environment variables or to assign the value of an environment variable to a CLIST or REXX variable. The syntax is as follows:

GETENV
prints the values of all environment variables.

GETENV *varname*
prints the value of *varname*.

GETENV *varname EXECvar*
stores the value of the environment variable *varname* in the CLIST/REXX variable *EXECvar*, or returns nonzero if the environment variable is not defined.

The PUTENV Command

The PUTENV command is used to assign a new value to an environment variable. If the variable does not exist, it is created. If it does exist, the old value is replaced. Note that the same variable name can be defined as both PERMANENT and EXTERNAL, in which case, they are different variables. Only the PERMANENT value is retained at the end of the session. The syntax for the PUTENV command is as follows:

```
PUTENV name=value[scope]
```

The *value* argument is assigned to *name*. *scope* may be either PERMANENT or EXTERNAL; if *scope* is omitted, EXTERNAL is assumed. (See Chapter 4, "Environment Variables," in SAS/C Library Reference, Volume 1 for information about environment variable scopes.)

Syntax notes:
In addition to the syntax shown, the PUTENV command allows you to omit the equal sign (=) (so long as there is a space between the name and the value) or to separate the equal sign (=) from the name and the value using spaces. An environment variable value containing blanks may be specified by enclosing the value string in double quotes. Finally, note that if an equal sign (=) is present, the value can be omitted, in which case the environment variable is assigned a null value.

Accessing PUTENV and GETENV via the CALL command

If your site has not made the GETENV and PUTENV commands available as TSO commands, you can still access them with the TSO CALL command as follows:

CALL 'SASC.TSOLOAD(*command_name*)"*operands*'

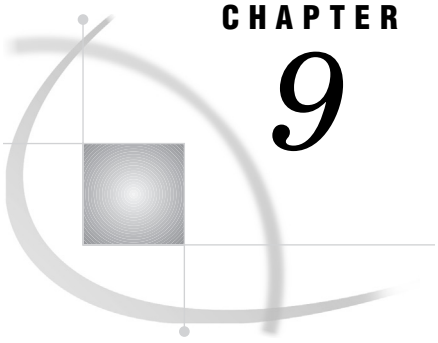
command_name is either GETENV or PUTENV, and the *operands* are as described earlier. The following shows a command-line example, followed by that same command executed with the CALL command:

```
PUTENV  _LOCALE=DBCS

CALL 'SASC.TSOLOAD(PUTENV)'  '_LOCALE=DBCS'
```

When you use the CALL command to invoke the PUTENV command, you must include the equal sign (=) between the variable name and the value, or the command

will be interpreted as a GETENV command. Also, because the CALL command uppercases its arguments, it cannot be used to assign values containing lowercase letters, unless you use the ASIS keyword of the CALL command. Your site may have changed the name of the SASC.TSOLOAD file. Check with your administrator to determine the name at your site.



CHAPTER 9

Run-Time Argument Processing

Introduction

This chapter explains how program arguments can be transferred from the external environment to the program. The program can receive arguments by accessing the **`argv`** or **`argc`** parameters to **`main`** or by environment variables.

This chapter also covers how to select certain run-time library parameters, as well as redirection of the files **`stdin`** and **`stdout`**.

Types of Run-Time Arguments

When you run a C program under CMS or the UNIX System Services (USS) shell, data are passed to the C program and the library as the part of the command line that follows the command name; the command name is used as the program name and is passed in **`argv[0]`**. This is also true in TSO if you use the optional support for invoking a C program from the DDname CPLIB. (See Chapter 8, “Executing C Programs,” on page 171.) If you use the TSO CALL command or OS/390 batch JCL to invoke a C program, the PARM string corresponds to the command line. The command line

normally consists of a number of tokens separated by space such as blanks, tabs, and so on. The tokens fall into these five classes:

- environment variable assignments, which have the form **=x=y**
- library options, which have the form **=x**
- I/O redirections, which begin with **<** or **>**
- argument redirections, which have the form **=<***filename*.
- program arguments, which include any other tokens.

Tokens of the various types can be intermixed in the command line.

Arguments containing blanks can be passed to the program by enclosing the argument in quotation marks, as in the following example:

```
"abc "
```

The normal C escape sequences such as \0 and \n can be used within quoted tokens. (Octal or hexadecimal escape sequences cannot be used.) On terminals without a backslash character (\), the cents sign (¢) can be substituted for the backslash.

Run-time library options and redirections can also be defined in the program source code. Subsequent sections provide details on how to do this.

Environment Variables

Environment variables are used to pass variable values that are set outside the program to be passed into the C program. An environment variable assignment begins with an equal sign (=) and contains one other equal sign either embedded or trailing, as in the following example:

```
=name=value
```

Environment variables are normally accessed using the standard **getenv** function or the POSIX variable **environ**. Alternately, you can define a third argument to **main** that is an array of pointers. You can access the values specified for environment variables by stepping through this array until you reach an argument that contains a null pointer. For example, you can specify the TSO or CMS command line

```
pgm =ABC=DEF =GHI=JKL =PATH=C:/LC
```

or the OS/390 PARM string

```
//STEP EXEC PGM=pgm,PARM='=ABC=DEF =GHI=JKL =PATH=C:/LC'
```

to invoke the following program:

```
void main(int argc, char **argv, char **envp)
{
   int i;
   i=0;
   printf( "program name is %s\n", *argv );

      /* Obtain program arguments. */
   while(--argc > 0)
      printf("argv[%d]  is %s\n", ++i, *++argv);

   i=0;
      /* Obtain environment variables. */
   while(*envp)
      printf("envp[%d]  is %s\n", i++, *envp++);
```

```
}
```

The program prints the following:

```
program name is pgm
envp[0]  is ABC=DEF
envp[1]  is GHI=JKL
envp[2]  is PATH=C:/LC
```

If an environment variable assignment has no *value* (for example, =*name*=), any previously entered value for that environment variable is removed. For example, if you invoke the example program under CMS or in TSO with

```
pgm =ABC=DEF =GHI=JKL =ABC=
```

or under OS/390 with

```
//STEP EXEC PGM=pgm,PARM='=ABC=DEF =GHI=JKL =ABC='
```

the program prints the following:

```
program name is pgm
envp[0]  is GHI=JKL
```

The *value* portion of an environment variable assignment can be enclosed in parentheses. The parentheses are removed from the environment variable string before it is stored. Parentheses enclosing environment variable assignments should balance. If there are more left parentheses than right, the remainder of the command line is regarded as part of the environment variable value. If there are more right parentheses than left, those parentheses become program arguments. No error message is issued if parentheses do not balance.

The colon (:) is not valid in the *name* portion of an environment variable entered on the command line.

Note: Under CMS and in TSO, environment variables can be defined externally to the program. Under CMS, the GLOBALV command defines environment variables. In TSO you can use the SAS/C PUTENV command. (See "Using the GETENV and PUTENV TSO Commands" on page 182 for more information.) These environment variables are not stored in the **envp** array in either case. △

Also note that when you run a SAS/C program under the USS shell, it inherits environment variables exported from the shell as well as any specific to the command line. Both sets of variables are stored in the **envp** array.

POSIX Considerations

For historical reasons, SAS/C environment variable support prior to Release 6.00 did not treat environment variable names as case-sensitive. That is, **getenv("sauce")**, **getenv("Sauce")**, and **getenv("SAUCE")** always returned the same result.

With Release 6.00, POSIX support has been added as described in Chapter 19, "Introduction to POSIX," in SAS/C Library Reference, Volume 2. The POSIX standards do not permit this case-insensitive behavior. For this reason, the names of environment variables in a program invoked with **exec**-linkage are considered case sensitive. Programs running in other environments, for example, TSO, retain the old behavior.

In the TSO environment, SAS/C supports several scopes of environment variables: program scope, external scope, and permanent scope. When a TSO program, or a child of a TSO program that was created by a call to the **fork** function, invokes an **exec** function, only the program scope variables are passed to the executed program. Note that although the library performs TSO environment variable name comparisons without reference to case distinctions, it preserves the case of characters in the

environment variable name for accurate transmission by the **exec** function. For instance, if a TSO program calls **putenv("Sauce=Bernaise")** followed by an **exec**, the new process will receive the environment variable Sauce, not sauce or SAUCE.

SAS/C POSIX support also removes the previous limitations on the size of program scope environment variable names and values. However, these limitations still apply to external and permanent scope variables.

The values of program scope environment variables can be inspected by the program without calling **getenv**. For programs compiled with the **posix** option, the environment variables are chained from the **extern char **environ**, as required by the POSIX.1 Standard. For programs not compiled with the **posix** option, the variable name **environ** is not reserved (as that would be a C Standards violation). However, there is a SAS/C **extern char ***_environ**. The pointer ***_environ** always addresses the program scope environment variable chain, regardless of whether the program is compiled with or without the **posix** option.

Updates to the program scope environment variables can be performed using the functions **putenv**, **setenv**, and **clearenv**. (**setenv** and **clearenv** are defined by the POSIX.1a Standard; **putenv** is a SAS/C extension.) Note that alteration of **environ** to change the program's environment variables should not be attempted, since portable functions to do this are provided.

As nonstandard extensions, SAS/C still supports the specification of environment variable values on the command line (even for programs invoked by the USS shell). It also supports supplying a third argument for the **main** function to receive a pointer to the program scope environment variables.

Run-Time Options

This section covers the various run-time options you can specify for program execution. These options can be specified in the following two ways:

- on the program command line
- in the program source code.

Although both forms of the option control the library, options specified on the command line are usually referred to as *command-line* options, and options coded in the program source are referred to as *program-specified* options. No matter which method you use to specify options, there is a limit of 1000 characters for the string that contains run-time options and user parameters. Note that it is possible to circumvent this limit using argument redirection.

Command-line options have the form **=*option*** . Options specified in this way are only in effect for the current execution of the program.

Program-specified options are specified by an external variable and are in effect for every execution of the program (unless overridden by an option from the command line).

For example, under CMS the **=minimal** command-line option is specified on the program's command line as follows:

```
cpgm arg1 =minimal
```

Specifying the option in this form indicates that you want a minimal form of program linkage for only this execution of your program. If you wanted always to have the **minimal** option in effect, however, you would code it in the program as follows:

```
char _linkage = _MINIMAL;
```

Note that program option specification is not portable.

The run-time options can be grouped as follows:

- general options

- linkage options
- memory allocation options
- program-only options.

Each group of options is discussed in detail in the following sections.

General Run-Time Options

These options control basic actions performed by the SAS/C Library. Most of these options can be specified either directly on the command line or in the program source. Command-line options begin with the equal sign (=). Most command-line options can be negated using the prefix **no**. For example, the negation of the **=warning** option can be specified as **=nowarning**.

Command-line options can be abbreviated by omitting final characters if no confusion with another option is possible. The abbreviation for each option is listed later in this section with the option description. Options can be specified in either upper- or lowercase.

Each option has a default specification. A program also can specify its own default by initializing an external variable for the option. A command-line specification always overrides any program-specified default.

Table 9.1 on page 189 lists sets of general options available for controlling run-time processing. Each option is discussed in detail following the table. The command-line specification is covered in the first section, followed by the program-specification form. General run-time options specified in the program source must be in uppercase.

Table 9.1 General Run-Time Options

Specified on the Command Line		Specified in the Program Source	
option	negation	int_options	int_negopts
=abdump	**=noabdump**	**--ABDUMP**	**_NOABDUMP**
=btrace	**=nobtrace**	**_BTRACE**	**_NOBTRACE**
=cnftrace	**=nocnfrace**	**_CNFTRACE**	**_NOCNFTRACE**
=debug	**=nodebug**	**_DEBUG**	**_NODEBUG**
=fillmem	**=nofillmem**	**_FILLMEM**	**_NOFILLMEM**
=hcsig	**=nohcsig**	**_HCSIG**	**_NOHCSIG**
=htsig	**=nohtsig**	**_HTSIG**	**_NOHTSIG**
=multitask	**=nomultitask**	**_MULTITASK**	**_NOMULTITASK**
=quit	**=noquit**	**_QUIT**	**_NOQUIT**
=storage	**=nostorage**	**_STORAGE**	**_NOSTORAGE**
=usage	**=nousage**	**_USAGE**	**_NOUSAGE**
=version	**=noversion**	**_VERSION**	**_NOVERSION**
=warning	**=nowarning**	**_WARNING**	**_NOWARNING**

Specified on the Command Line		Specified in the Program Source	
option	negation	int_options	int_negopts
=xtrace	**=noxtrace**	**_XTRACE**	**_NOXTRACE**
=zeromem	**=nozeromem**	**_ZEROMEM**	**_NOZEROMEM**

The meaning of each general run-time option is discussed below. The options are shown in command-line form; the program specification is discussed following this section.

=abdump
=a
produces a dump when an ABEND occurs, including an ABEND that is recovered by a **SIGABRT** or **SIGABND** signal handler. Under OS/390, **=abdump** is only meaningful in the case that an ABEND is recovered, since a dump is always produced (if an appropriate DD card is allocated) if the ABEND is not recovered.

Obtaining an ABEND dump under the USS shell can be tedious, because the mechanism that supports this (defining a SYSMDUMP data set before invoking the TSO OMVS command to bring up the shell) applies to all descendants of the shell. The **=abdump** run-time option overcomes this difficulty. If **=abdump** is specified for a program with **exec**-linkage, the library allocates the OS/390 data set userid.SASC.DUMP to the DDname SYSUDUMP during program startup. If the file cannot be allocated (perhaps because it is in use by another process) a diagnostic is generated and execution proceeds normally. You can use a dump file other than userid.SASC.DUMP with the **=abdump** option if you export the environment variable **ddn_SYSUDUMP** with the name of an alternate dump data set.

=btrace
=b
causes a traceback to be included with library warning messages. The default is **=nobtrace**.

Note: The **=btrace** name is used to distinguish this option from the **=trace** option, which invokes the debugger. △

=cnftrace
displays TCP/IP configuration information. The display has the following format:

```
LSCX077 TCP/IP Config Trace: [text]
```

=debug
=d
requests the use of the SAS/C Debugger, as described in the SAS/C Debugger User's Guide and Reference.

If you are executing an all-resident module using the **=debug** or **=trace** options, you must also define the macro name **ALLOW_TRANSIENT** in the module that includes **<resident.h>**.

Note: When you specify this option, **=fdump** option is also in effect. **=fillmem** is in effect unless it is overridden. The default is **=nodebug**. △

=fillmem
=fi
specifies that when memory is allocated by the run-time library, it should be filled with the fill character 0xfc. This causes uninitialized variables to have unusual values (for example, an uninitialized **int** has a value of approximately —48,000,000), which increases the chance that the error will be detected. The

default is **=nofillmem**, unless the debugger is used. When the debugger is used, **=fillmem** is the default.

Note: The use of **=fillmem** forces the use of the **=fdump** linkage option. Also, the use of **=fillmem** substantially increases execution time. △

=hcsig
=hc
specifies that the library is to intercept computational signals (**SIGFPE**, **SIGSEGV**, and **SIGILL**) using SPIE or ESPIE. The default is **=hcsig**. Specifying **=nohcsig** prevents library handling of these signals. The following characteristics also pertain to **=hcsig**:

- When **=nohcsig** is in effect, these signals cause an immediate abend; therefore, handling of overflow and underflow by library mathematical routines is impossible in some cases.
- When the debugger is used, **=nohcsig** also applies to the debugger; therefore, the debugger features for recovery from invalid pointer usage are ineffective.
- If **=nohcsig** and **=htsig** are used together, the SPIE or ESPIE macro is still used during ABEND handling in order to protect the ABEND exit from errors caused by corruption of CONTROL blocks. However, the SPIE or ESPIE in effect at the time of the ABEND is always restored by the library's traceback routine before it returns control to the system.

=htsig
=ht
specifies that the library is to intercept program termination signals (**SIGABRT** and **SIGABND**) using ESTAE under OS/390 or ABNEXIT under CMS. The default is **=htsig**. Specifying **=nohtsig** prevents library handling of these signals.

Note: When **=nohtsig** is in effect, no traceback can be produced at program termination. Also, when the **system** function is used to call a TSO command, ESTAE is always used to protect TSO from the effects of an ABEND during this processing. This exit produces no messages and is in effect only during execution of the **system** function. Use of **=nohtsig** should seldom be required. The library's ABEND handling routine is written so that it does not interfere with other ABEND exits established by the caller of the C program or by assembler routines called from C. △

=multitask
=mu
specifies an alternate implementation of communication between a C program and the debugger or between a C program and other high-level languages. **=multitask** helps isolate the programs and languages from each other and reduces the chances that an error in one will cause the other to fail. However, use of **=multitask** causes additional overhead, and it is more suited for use during program development than in a production program. The default is **=nomultitask**. Refer to the SAS/C Compiler Interlanguage Communication Feature User's Guide for more information on the **=multitask** option.

Under OS/390, **=multitask** causes the C program, the debugger, and each non-C language to run as a separate task. The **fork** library function may not be used if **=multitask** is in effect.

=quit
=q
causes program execution to be abnormally terminated after any library warning message is generated. This can be useful for obtaining a dump in such a situation. If warnings are suppressed by the **quiet** function, the **=quit** option has no effect unless the **=warning** option is also specified. The default is **=noquit**.

=storage
=s
causes the library to create a storage analysis report at program termination or in the event of an ABEND. The report is identical to the output of the debugger's storage command. See the SAS/C Debugger User's Guide and Reference for detailed information about how to use the report.

If you are executing an all-resident module using the **=storage** option, you must also define the macro name **ALLOW_TRANSIENT** in the file that includes **<resident.h>**.

Under CMS, the report is written to STGRPT LISTING A1. In TSO or under OS/390 batch, the report is written to DDname STGRPT. Under the USS shell, the report is written to the file **storage.out** in the current directory. If the report is created at normal program termination, it is titled "Normal Termination Storage Report." If it is created during abnormal termination, it is titled "Abnormal Termination Storage Report."

=usage
=u
causes a storage usage report to be printed at program termination. This report can be used to determine the required stack size for a program that is to use the **=minimal** option. The default is **=nousage**.

=version
causes the release numbers associated with the resident and transient libraries to be displayed at program start-up. The display has the following format:

```
LSCX056 SAS/C library release n.nnx  (resident)
                      release n.nnx  (transient)
```

where n.nnx is the release number. The release number of the resident library does not have to be the same as the release number of the transient library. This information can be helpful in determining library mismatches. See Chapter 1, "Introduction to the SAS/C Library," in SAS/C Library Reference, Volume 1 for information about using different releases of the Compiler and Library.

=warning
=w
forces library warning messages to be printed even if the **quiet** function is used by the program to suppress them. The **=warning** option also generates a traceback even when the program is cancelled, as by an OS/390 operator cancel or a CMS HX command. The traceback is ordinarily suppressed in this situation. This option may be useful for getting diagnostic information about a looping program. The default is **=nowarning**.

=xtrace
=x
causes a diagnostic to be printed, including a traceback, whenever a C++ exception is thrown. Only one traceback is printed for an exception, even if the exception is caught and rethrown.

=zeromem
=z
specifies that when memory is allocated by the run-time library, it should be filled with 0s. This causes uninitialized variables to be set to 0, which may allow erroneous programs to execute successfully. The default is **=nozeromem**.

Note: Using **=zeromem** forces the use of the **=fdump** linkage option. Also, using **=zeromem** substantially increases execution time. If both **=fillmem** and **=zeromem** are specified, **=fillmem** is ignored. △

Program specification

To specify general run-time options in your program, initialize the integer variable **_options** with one or more bit flags to specifically request one or more options. The **_options** variable must be an external variable.

Include **<options.h>** to obtain the names of the flags for assignment to **_options**. The currently implemented options are as follows:

_ABDUMP	produces a dump when an ABEND occurs.
_BTRACE	prints a traceback with each diagnostic.
_DEBUG	invokes the debugger.
_FILLMEM	fills the memory with 0xfc when allocated.
_HCSIG	handles computational signals.
_HTSIG	handles abnormal termination signals.
_MULTITASK	uses a multitasking debugger interface.
_QUIT	terminates execution after a diagnostic.
_STORAGE	produces storage corruption report after execution.
_USAGE	prints a storage usage report after execution.
_VERSION	prints the library version number.
_WARNING	always prints the run-time warning messages.
_ZEROMEM	indicates zero memory when allocated.

For example, the following code assigns flags to the **_options** variable:

```
extern int _options = _BTRACE + _WARNING;
extern int _negopts = _NOHCSIG;
```

You can initialize the integer variable **_negopts** to reset (turn off) one or more options. Do not specify the same option for both the **_options** and **_negopts** variables; if you do, the result is undefined.

Include **<options.h>** to obtain the names of the flags for assignment to **_negopts**. The following are currently implemented options:

_NOABDUMP	does not produce a dump when an ABEND occurs.
_NOBTRACE	does not print a traceback with each diagnostic.
_NODEBUG	does not invoke the debugger.
_NOFILLMEM	does not fill the memory with 0xfc when allocated.
_NOHCSIG	does not handle computational signals.
_NOHTSIG	does not handle abnormal termination signals.
_NOMULTITASK	does not use a multitasking debugger interface.
_NOQUIT	does not terminate execution after a diagnostic.
_NOSTORAGE	does not produce storage corruption report after execution.
_NOUSAGE	does not print storage usage report after execution.
_NOVERSION	does not print library version number.
_NOWARNING	does not force printing of run-time warning messages.
_NOZEROMEM	does not indicate zero memory when allocated.

For example, the following code prevents the library handling of computational signals:

```
int  _negopts = _NOHCSIG;
```

Linkage Run-Time Options

Linkage options are a subset of the library options that specify which prolog and epilog code should be executed with your program. Even after link-editing your program, you have some flexibility choosing which prolog and epilog code is executed at function entry and return. The choice of linkage option can affect considerably how fast the program executes and how easy it is to debug. Note that the linkage options cannot be negated. The **=inter** option is the default linkage option.

Table 9.2 on page 194 summarizes the linkage run-time options. The linkage options and their effects follow the table.

Table 9.2 Linkage Run-Time Options

Specified on the Command Line	Specified in the Program Source
=fdump	**_FDUMP**
=inter	**_INTER**
=minimal	**_MINIMAL**
=optimize	**_OPTIMIZE**

=fdump
=fd
specifies that you want dump formatting support. This option is the most expensive and significantly increases function call overhead. However, it fully implements normal save-area chaining conventions and labels each save area with the name of the corresponding function, thereby improving dump readability.

=inter
=i
specifies that support for communication with assembler language, or other non-C code, is required. This is the default specification. In addition to providing interlanguage communication support, this option improves the reliability of the library's abend traceback. Use of this linkage option is recommended during program testing.

=minimal
=mi
specifies that a minimal form of program linkage is desired. Use of minimal linkage is recommended only for thoroughly tested and reliable programs for which performance is critical. Refer to **=usage** in "General Run-Time Options" on page 189 and to ***=nnn/mmm*** in "Memory Allocation Options" on page 195 for information on obtaining the stack size to specify when you use the **=minimal** option.

When minimal linkage is requested, a single area of memory is allocated for automatic storage when the program starts up; overflow of this area is not checked. If overflow occurs, random abends or overlays of the program or other data are to be expected.

Note: Do not use this form of linkage for programs using recursive algorithms unless you know the upper bound to the amount of recursion required. △

Use of this option minimizes the overhead of function calls, producing significant savings. However, all of the restrictions described for optimized linkage also apply to minimal linkage. Also note that due to the difference in automatic storage layout when **=minimal** is specified, storing data outside the bounds of an array is more likely to overlay other data.

=optimize
=o

specifies that you want an optimized form of program linkage. When this option is specified, function call overhead is decreased. However, the following restrictions must be observed. If they are not observed, the effects are unpredictable.

- Only C and assembler language subroutines are permitted.
- When assembler subroutines are used, these routines must not call C functions unless the assembler CENTRY and CEXIT macros are used.
- Functions compiled with the **indep** option cannot be used.
- Program checks that occur in assembler language routines cannot be handled by the C program.

Optimized linkage is recommended for production programs that meet the restrictions described above.

Note: In some cases, use of optimized linkage may prevent the generation of an accurate traceback on abnormal termination. △

Program specification

You can initialize the character variable **_linkage** to specify a linkage option. Include **<options.h>** to obtain the names of the values for assignment to **_linkage**. The following are the currently implemented linkage options:

_FDUMP	supports dump formatting.
_INTER	supports linkage with other languages.
_MINIMAL	suppresses stack overflow checking.
_OPTIMIZE	supports optimized linkage.

The following is an example of assigning the **_FDUMP** option to the **_linkage** character variable:

```
extern char _linkage = _FDUMP;
```

Memory Allocation Options

A run-time option can be specified on the command line to request the initial stack or heap allocation size, or both. The syntax of this option is as follows:

=nnn/mmm

nnn is the starting stack size and ***mmm*** is the starting heap size. (The sizes can be expressed either as integers or as integers followed by an upper- or lowercase K.) To specify a stack size only, use **=*nnn*** (omit the slash). To specify a heap size only, use **=/*mmm*** (include the slash). The default heap size is 4K. The default starting stack size is 4K unless **=minimal** is specified, in which case the default is 32K.

Note: If the program initializes the external variable ***_mneed*** to a non-zero value to indicate the use of the obsolete **sbrk** function, the ***mmm*** portion of the statement is interpreted as the size of the **sbrk** area rather than as the size of the heap. △

The following are examples for each operating system of commands that specify several run-time options:

- Under CMS:

```
exam =o nolist =/80k =noht "sep(' ')"
```

- In TSO:

```
call (exam) '=o nolist =/80k =noht "sep('' '')"' asis
```

- Under the USS shell:

```
exam =o nolist =/80k =noht "sep(' ')"
```

- Under OS/390:

```
// EXEC PGM=EXAM,
// PARM='=o nolist =/80k =noht "sep('' '')"'
```

Each example invokes the program **exam**. The library options **=optimize** and **=nohtsig** are specified, and an initial heap allocation of 80K is requested. Two arguments, which have the values **nolist** and **sep(' ')**, are passed to **exam**.

Program specification

These memory allocation options can be specified in the program as follows:

```
int _stack   = value;
int _heap    = value;
int _mneed   = value;
```

You can initialize the integer **_stack** to a numeric value to request a specific initial allocation of stack space.

You can initialize the integer **_heap** to a numeric value to request a specific initial allocation of heap (**malloc**) space.

You can initialize the integer **_mneed** to a numeric value to request a specific initial allocation of **sbrk** space.

The following is an example of how to force a large heap allocation:

```
int     _heap = 1024000;
```

Note that you can override a **_heap** or **_mneed** specification by using the **=nnn/mmm** option for the command line.

Program-only Options

This section describes several additional external variables that can be initialized by the program to request special library processing. These options are available only by using these external variables; that is, they cannot be specified at execution time via the command line.

You can initialize the integer variable **_nio** to any non-zero value to indicate that the C program performs no I/O with C library routines. The overhead of opening the standard files and loading I/O routines can thus be avoided. For example:

```
int  _nio      = nonzero ;
```

You can initialize the integer variable **_nlibopt** to any non-zero value to suppress the use of library options or redirection on the command line. If **_nlibopt** is set, all

tokens on the command line are passed to the program, even if they resemble run-time options. For example:

```
int  _nlibopt  = nonzero ;
```

You can initialize the pointer **_pgmnm** to the address of a string literal to be used as the program name when no name can be obtained from the operating system. For example:

```
char *_pgmnm   = string ;
```

stkabv and stkrels External Compiler Variables

Older versions of OS/390 were limited to running with 24-bit addresses, giving a maximum virtual address space of 16 megabytes. With the release of MVS/XA the addresses were increased to 31 bits giving a virtual address space maximum of 2 gigabytes. Certain portions of OS/390 (notably certain I/O subsystems) were not modified to accept 31-bit addresses, therefore programs wishing to utilize these services were forced to get storage below the 16M line to use as parameters when calling these functions. Prior versions of SAS/C allocated all stack memory from the area below the line to avoid the problems involved in calling old OS/390 services with 31-bit addresses.

In SAS/C Release 6.50, defining the external integer variable **_stkabv** in the source program (example: **extern int _stkabv = 1;**) will indicate to the library to allocate stack space above the 16M line.

Note: Setting the variable at run time will have no effect; it must be *initialized* to 1 as shown. △

However, some SAS/C library functions require their stack space be allocated below the line due to their use of auto storage for parameter lists and control blocks which still have a below-the-line requirement. These library routines have been identified, and either modified to remove the requirement, or changed to request that their own allocation of stack space be located below the 16M line. Release 6.50 includes a compiler option (STKBELOW) and a **CENTRY** macro parameter (STKBELOW=YES) to allow user code to request that its stack space be allocated below the line even if the **_stkabv** variable is defined as non-zero.

A new option allows the library to release stack space that is no longer needed. To free stack space, define the external integer variable **_stkrels** (example: **extern int _stkrels = 1;**). This tells the library that, on return from a function, if an entire stack segment becomes unused, the segment should be returned to the operating system. This option is useful in long running programs that contain code paths that can occasionally become deeply nested, or in multi-tasking applications. Use of **_stkrels** and **_stkabv** guarantee that no stack space is allocated below the line if none is required by an executing routine.

Interleaved SYSOUT

Multi-tasking applications that share SYSOUT data sets between tasks may experience ABENDs when two or more tasks attempt to write to SYSOUT at the same time. A new external variable, **_isysout**, has been provided to prevent this type of ABEND.

_isysout
: defining the external integer variable **_isysout** in the source program (for example, **extern int _isysout = 1;**) indicates to the library that access to data

sets with a SYSOUT type must be serialized. ABENDs may occur in a multitasking program (notably, ABEND S02A, reason x'0C') when more than one task attempts to write to a SYSOUT data set at the same time. The **_isysout** variable must be initialized to 1 in each task of a multitasking SAS/C application that could potentially write to a common SYSOUT data set.

This support is for OS/390 only, and OS/390 Name/Token services must be available at run time.

Standard File Redirection

Redirections allow the choice of the standard input and output files to be made at run-time. Redirections are affected by the **_style** variable. The **_style** variable must be an external variable.

The **_style** variable can have the value of any valid pathname style and determines the default style prefix. For example, assume you have a program containing the following declaration:

```
char *_style = "tso";
```

Then, the following TSO program call would request that **stdin** be opened to **tso:input.data(file5)**:

```
call mypgms(simple) '<input.data(file5)'
```

Refer to the discussion of **_style** in "Program Standard File Specification" on page 199 for additional information.

Alternate CMS Redirection Format

The library under CMS allows two forms of fileid specification in a redirection. The usual specification is of the following form:

```
simple -x -y <data.file.a >output.listing
```

The fileid is simply any valid CMS pathname in the no-blanks format. Another example of this form is the following:

```
simple -x -y <cms:rdr >printer
```

This form of fileid specification is subject to truncation, however, if only a CMS tokenized parameter list is available to the library. If the C program can be invoked so that only a CMS tokenized parameter list is available to the library, you should use the alternate form of redirection. In this alternate form, the filename, filetype, and filemode are specified as separate tokens and surrounded by parentheses, as in the following:

```
simple -x -y <(data file a) >(output listing)
```

or

```
simple -x -y <(rdr) >(printer)
```

This form of fileid is not subject to truncation in a tokenized parameter list.

Program Standard File Specification

A program also can specify an input or output redirection for standard C files. Options controlling standard file redirection can be specified in the program source as follows:

```
char *_stdinm  = string ,
     *_stdonm  = string ,
     *_stdenm  = string ;

char *_stdiamp = string ,
     *_stdoamp = string ,
     *_stdeamp = string ;
```

The **_std***xxxx* variable must be an external variable. The variables are used as follows:

- You can initialize the pointer **_stdinm** to the address of a string literal to name a specific file to be opened as **stdin**.
- You can initialize the pointer **_stdonm** to the address of a string literal to name a specific file to be opened as **stdout**.
- You can initialize the pointer **_stdenm** to the address of a string literal to name a specific file to be opened as **stderr**.
- You can initialize the pointer **_stdiamp** to the address of a string literal to define access-method parameters (amparms) to be used by the library when opening **stdin**. This variable can be used to control whether the program name is used as a prompt or whether any prompt is issued at all. For example, the following statement suppresses the prompt:

  ```
  char *_stdiamp = "prompt=";
  ```

  The following statement causes the prompt "Enter command" to be issued whenever **stdin** is accessed:

  ```
  char *_stdiamp = "prompt=Enter command\n";
  ```

- You can initialize the pointer **_stdoamp** to the address of a string literal to define access-method parameters to be used by the library when opening **stdout**.
- You can initialize the pointer **_stdeamp** to the address of a string literal to define access-method parameters to be used by the library when opening **stderr**.

You can also initialize the character pointer variable **_style** to address a string that will be used by the run-time system as a style prefix for all I/O open requests that do not specify a style prefix. However, note that **_style** is ignored for a program compiled with the **posix** option.

The format for **_style** is as follows:

```
char *_style   = string ;
```

The string should be no longer than four characters (not including the optional final colon and the terminating null). The value specified for **_style** affects filenames specified on the command line. For example, under CMS if **_style** is set to **ddn:** and PRINTER is specified on the command line, output is sent to DDname PRINTER and not to your virtual printer.

Note: Program option specification is not portable. △

For more information on filenames and styles, and amparms, refer to Chapter 3, "I/O Functions," in SAS/C Library Reference, Volume 1.

Argument Redirection

The command-line token **=<***filename* defines an argument redirection. An argument redirection is processed by opening the file specified, reading it in its entirety, and replacing the **=<***filename* token with the file contents. The file can contain environment variable assignment, library option, or program argument tokens. It can also contain additional argument redirections. However, it cannot contain I/O redirections.

Argument redirection can be used to insert tokens in a command-line argument string. For instance, if the command line specifies the following:

```
arg1 =<tso:myargs arg2
```

and the TSO file userid.MYARGS contains the following:

```
arg3 arg4 =ENV1=22 =WARN
```

then the program is called as if the command line specified

```
arg1 arg3 arg4 =ENV1=22 =WARN arg2
```

Rules for Using Argument Redirection

Here are some more specific rules about the use of argument redirection:

1. An argument redirection file name is interpreted as if it were a filename passed to **`fopen`**. In particular, if no style prefix is present in the name, the default style specified by the program using **`_style`** applies.
2. Under CMS, the notation **=<(***filename filetype filemode***)** can be used to handle filenames containing blanks.
3. New lines and null characters in the argument redirection file are replaced with blanks.
4. You cannot use the terminal or a pipe as an argument redirection file.
5. An argument redirection file should not contain an input or output redirection or a stack/heap size specification. These can only appear on the command line proper. If either of these is found in an argument redirection file, it is ignored.
6. Recursive use of argument redirection (that is, an argument redirection file that directly or indirectly respecifies itself) is detected. The same file may be included twice under different names, but an infinite loop recursively reading the same file will always be avoided.
7. Although the length of the actual command line is limited by the SAS/C Library to 1000 characters, there is no limit to the size of the command line constructed after all the argument redirections have been processed. Thus, argument redirection can be used to circumvent this library limitation.
8. Argument redirection is supported in the run-time options string passed to $MAINO and other alternate C start-up routines. However, in this case, any program arguments (as opposed to run-time options or environment variable assignments) in an argument redirection file will be ignored.
9. An important use of argument redirection is to specify more than one or two environment variables in batch, where the PARM string is limited to 100 characters.

CHAPTER

10

All-Resident C Programs

Introduction

Normally, when a C program is linked, the resulting load module does not contain all of the support routines needed by the program. For example, before the program's **`main`** function is entered, the command line must be parsed and the **`argv`** vector created. Because the command-line parsing routine is only needed once, during program start-up, the program initialization routine dynamically loads it from the transient library and unloads it (freeing the memory it required as well) when it is no longer needed.

In most programming situations, the dynamic loading and unloading of support routines makes the best use of available resources. User storage is not occupied by unused code, and when the support routines are installed in shared memory, many users can access a single copy of the routine. Also, the load module is much smaller because it contains only a small percentage of the required code.

However, 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 be used. The following sections describe how to create all-resident programs.

The rest of this chapter is divided into five sections. The first section describes the organization of the all-resident library, that is, what each collection of support routines is for and how they are grouped. The second section shows how the **`<resident.h>`** header file can be used to specify which support routines are needed by an all-resident program. The third section lists some restrictions that apply to all-resident programs.

The fourth section discusses several programming considerations. The last section tells you where information about linking all-resident programs can be found in this book.

Throughout this chapter, support subroutines are referred to as routines rather than functions in order to prevent confusion with library functions such as **strcpy**.

Library Organization

It is important to remember that many functions in the C library do not depend on transient support routines. String functions such as **strcmp** and math functions such as **cos** are examples of such functions. Complex functions or functions that interact closely and frequently with the operating system are likely to be transient. For instance, I/O functions such as **fopen** and signal-handling functions such as **alarm** are implemented as calls to transient support routines.

These library support routines do not necessarily form a one-to-one correspondence with the calling function. Often, support routines can be shared among several callers, and closely related routines can be packaged together. For example, **fopen** and **afopen** share some support routines, and all of the support routines for performing I/O to VSAM data sets are packaged as a unit.

Another important point to remember is that it is not possible to identify exactly, at link time, the smallest set of support routines required by a given program. For example, the linker can be relied upon to include the **strstr** function only if the program contains an external reference for the function, but it is impossible to determine that no VSAM files will be opened by examining the external reference for **fopen**. In many cases, even the programmer cannot predict which support routines will be required.

The <resident.h> Header File

Given the large number of available library functions and the wide variety of functionality they provide, it is probable that few C programs make use of all the library support routines. Even though it is typically very difficult to determine the minimum number of required support routines, it is usually possible to specify some subset of the entire library. (For example, programs that do not use coprocesses do not need the coprocess support routines.) Therefore, an important part of the linking process must be some way for the programmer to specify the set of support routines required by the program. This tailoring ability is available via the **<resident.h>** header file.

Including the **<resident.h>** header file causes the compiler to generate external symbols for various groups of support routines. Each external symbol causes a corresponding group of support routines to be included when the program is linked. Which external symbols are generated and, therefore, which support routines are linked into the program, is controlled by defining specific macro names prior to the inclusion of **<resident.h>** in the C source file.

For example, defining the macro name **RES_VSAM** causes **<resident.h>** to generate an external symbol for the VSAM I/O support routines when the file including it is compiled. When the program is linked (including the object code from the **<resident.h>** file), the linker includes these routines as part of the program load module.

By confining the all-resident tailoring process to a single source file, it is possible to create both an all-resident version of the program and a transient version. The all-resident version is created by including the **<resident.h>** object code and linking with the all-resident library. For the transient version, neither of these steps are necessary.

Note: An all-resident application should include **<resident.h>** in one and only one source module. Errors may occur if **<resident.h>** is included in several different modules of the same program. △

Identifying the Target Operating System

The **<resident.h>** header file generates external symbols that are specific to CMS, OS/390, or CICS. You can define the macro name **SYS_CMS**, **SYS_OSVS**, or **SYS_CICS** to target a specific system. If you do not define one of these macro names, **<resident.h>** tests the macro names **OSVS** and **CMS**, one of which is automatically generated by the compiler for each of those two operating systems to determine whether or not to generate symbols for OS/390 or CMS. If you are developing an all-resident application for CICS, you must define the macro name **SYS_CICS** before including the **<resident.h>** header file in your program.

Selecting the Routines to Be Included

Table 10.1 on page 203 shows the macro names that can be defined to cause inclusion of library support routines and describes the associated support routines. By default, these routines are not included in the program load module. The Operating System column indicates whether the support routines are used under OS/390, CMS, CICS, or all three. A number appearing in the Notes column indicates that the support routines are included automatically if certain functions are used in the program. An explanation of the numbers used in the Notes column follows Table 10.1 on page 203. (Refer to "Subordinate Load Modules" on page 207 for more information on when support routines are included automatically.)

Table 10.1 Macro Names Used with <resident.h> for Inclusion

Macro Names	Operating System	Notes	Includes Support for
RES_SIGNAL	all	1	signal-handling functions
RES_COPROC	all	2	coprocessing functions
RES_IOUTIL	OS/390, CMS	3	**remove, rename, access** functions; **cmsstat** function (CMS only)
RES_UNIXIO	OS/390, CMS	4	UNIX style I/O
RES_UNIXSTDF	OS/390, CMS		UNIX style I/O to **stdin, stdout,** and **stderr**
RES_TMPFILE	OS/390, CMS	5	VSAM I/O
RES_VSAM	OS/390, CMS	6	VSAM I/O
RES_DSNAME	OS/390		**dsn** and **tso** style filenames
RES_DIVIO	OS/390		DIV I/O
RES_TSOENVVAR	OS/390		TSO environment variable support
RES_VSAM_STDIO	OS/390, CMS		VSAM I/O using text or binary access
RES_KEYED_IO	OS/390, CMS		VSAM I/O using keyed access

Macro Names	Operating System	Notes	Includes Support for
RES_TCPIP	all	7	socket library functions
RES_HFS_STDIO	OS/390	8	UNIX System Services (USS) HFS I/O (plus **remove, rename,** or **access**)
RES_FDOPEN	OS/390	9	the **fdopen** function
RES_OE_SYSTEM	OS/390		**system** used to envoke a shell command
RES_SUBCOM	OS/390	10	TSO or USS SUBCOM
RES_FILEDEF	CMS		**ddn** style filenames
RES_OSSIM	CMS		OS/390-simulated I/O
RES_LIBIO	CMS		MACLIB and TXTLIB member I/O
RES_UNITREC	CMS		unit record I/O (virtual reader, printer, and punch)
RES_SHARED_FILE	CMS		CMS Shared File System I/O
RES_SPLFILE	CMS		CICS spool file I/O
RES_CICSVSE	CICS		VSE error handling
RES_FSSLSTD	OS/390, CMS		FSSL using the direct 3270 interface
RES_FSSLISPF	OS/390, CMS		FSSL using the ISPF interface

Note: The support routines are included automatically if

- the **signal** or **raise** function is used, or if signal support is needed for other reasons.
- the **costart** function is used.
- any of the **remove**, **rename**, **access**, or **cmsstat** functions are used.
- the **open** or **creat** function is used.
- the **tmpfile** function is used.
- **RES_VSAM** is defined. Support routines for any access (text, binary, or keyed) to VSAM files will be included. If **RES_VSAM** is not defined, defining **RES_VSAM_STDIO** will include support for text or binary access to VSAM files, and defining **RES_KEYED_IO** will include support for keyed access.
- any socket library functions are used. Note that the additional symbols **NO_OE_SOCKETS** and **ONLY_OE_SOCKETS** can be used in OS/390 to control which kinds of sockets are supported.
- you compile with the **posix** option.
- the **fdopen** function is used or you compile with the **posix** option.
- the **execinit** function is used. Note that the additional symbols **RES_SUBCOM_TSO** and **RES_SUBCOM_OE** can be used to include only TSO support or only USS support.

△

Selecting the Routines to Be Excluded

In addition to specifying which support routines should be included, **<resident.h>** respects a number of macro names that indicate that certain support routines should be

omitted. By default, these routines are included in the program load module, except for the subcom and socket routines. For an all-resident program that has specified **_nio** to be not equal to zero, the macro name **NO_IO** must be defined prior to including the **<resident.h>** header file in the program. Table 10.2 on page 205 shows these names and the associated support routines.

Table 10.2 Macro Names Used with <resident.h> for Exclusion

Macro Names	Operating System	Excludes Support for
NO_IO	all	C I/O functions
NO_WARNING	all	warning messages
NO_ABEND	all	ABEND handling
MVS370_ONLY	OS/390	MVS/XA ABEND handling
MVSXA_ONLY	OS/390	MVS/370 ABEND handling
MODE370_ONLY	CMS	XA-mode ABEND handling
MODEXA_ONLY	CMS	370–mode ABEND handling
NO_OE_SOCKETS	OS/390	nonintegrated sockets
RES_SUBCOMM_TSO	OS/390	USS SUBCOM support
RES_SUBCOM_OE	OS/390	TSO SUBCOM support

Using Dynamic Loading

<resident.h> allows the programmer to decide whether dynamic loading will be available to the program. There are three possible choices as follows:

- No dynamic loading is available. In this mode, neither the program nor the library can dynamically load another load module. The dynamic loading support routines are not included in the program load module. This is the default.
- The program itself uses dynamic loading, but the library cannot. In this mode, the program can use the **loadm** and **loadd** functions, but the library is prohibited from using them.

  If either the **loadm** or **loadd** function is used in the program, then the necessary support routines are linked with the load module automatically. Also, the macro name **ALLOW_LOADM** can be defined to indicate that the dynamic loading support routines are to be linked into the load module.
- Dynamic loading is allowed. In this mode, if a required support routine is not linked with the program, then it is loaded from the transient library. This mode is useful for situations in which certain support routines are used only rarely or while a program is under development and the set of required routines is still

volatile. More importantly, for programs under development, this mode is the only mode that allows the use of the debugger with all-resident programs because the debugger requires routines in the transient library. If this mode is selected, define the macro name **ALLOW_TRANSIENT**.

Using <resident.h>

Example Code 10.1 on page 206 is an example of using **<resident.h>**. In this particular use, signal-handling and UNIX style I/O support routines are linked with the program. Note that the header file and macro name definitions can be added to another program source file or confined to a source file by themselves.

Example Code 10.1 Sample Use of <resident.h>

```
#define RES_SIGNAL  /* Include signal handling support.        */
#define RES_IOUTIL  /* Include access, rename, remove support. */
#define RES_UNIXIO  /* Include UNIX style I/O support.         */
#define RES_TMPFILE /* Include temporary file I/O support.     */

#if defined CMS
#define RES_LIBIO   /* If CMS, include support routines for    */
#endif              /* I/O to MACLIB/TXTLIB members.           */

#if defined OSVS
#define MVSXA_ONLY  /* If OS, exclude MVS/370 ABEND handling.  */
#endif

#define ALLOW_LOADM /* Allow this program to load other load   */
                    /* modules, but no library modules can be  */
                    /* loaded.                                 */

#include <resident.h>
```

Note: The **dollars** compiler option must be used when compiling a C++ source file that contains **<resident.h>**. △

Restrictions

Some library functionality is restricted or unavailable when the all-resident library is used. The following restrictions apply to all-resident programs:

- The interlanguage communication feature described in the *SAS/C Compiler Interlanguage Communication Feature User's Guide* cannot be used. However, calls to and from assembler language programs are supported.
- The REXX function package support feature cannot be used. An all-resident program cannot call the **cmsrxfn** function.
- Socket support cannot be made resident unless the IBM TCP/IP product is used. If you need the ability to use TCP/IP support from a vendor other than IBM, you should avoid the all-resident library, or define the **ALLOW_TRANSIENT** symbol to allow the necessary code to be loaded at runtime.

- Under an extended architecture system, all-resident programs cannot be loaded above the 16-megabyte line. Programs must be linked with RMODE=24. However, 31-bit addressing (AMODE=31) is supported.
- Regardless of reentrancy considerations, all-resident programs must be processed with the COOL object code preprocessor.
- The **<resident.h>** header file should only be included in one source module per load module.

Development Considerations

The following items should be considered when developing all-resident programs.

Missing Support Routines

In a program that does not allow the library to dynamically load support routines, if a support routine is needed but has not been linked into the program load module, the library issues warning message LSCX119:

```
"Transient module name  could not be located."
```

Warning Messages

In a program that does not allow the library to dynamically load support routines, if a warning message is needed but has not been linked into the program load module, the library issues the generic message LSCX047:

```
"Unable to load runtime message texts, errno = EFORBID".
```

The **errno** value **EFORBID** indicates that the library cannot issue the correct message because **ALLOW_TRANSIENT** was not defined.

If the macro names **NO_WARNING** (see Table 10.2 on page 205) and **ALLOW_TRANSIENT** are defined in the source file containing **<resident.h>**, the library diagnostic message texts are not linked with the program but are loaded by the library, if needed. This combination may be helpful during program development.

Subordinate Load Modules

As mentioned above, it is possible for an all-resident program to dynamically load subordinate load modules even though the library is prohibited from doing so itself. The subordinate load modules use the support routines that have been linked into the primary load module. The reverse is not true, however. The primary load module cannot use support routines that are linked in a subordinate load module, even if the load module has been loaded into storage.

CAUTION:

Link support routines with the primary load module. In an application that uses subordinate load modules, always link the support routines with the primary load module. The subordinate load modules should be linked in the normal manner. △

If a function that usually causes the appropriate support routines to be included automatically (as indicated in Table 10.1 on page 203) is called only from a subordinate load module, then those routines are not linked with the load module. The primary load module must be linked with an object deck that is generated from a source file containing **<resident.h>** and that has the required symbol defined.

For example, suppose a program contains two load modules, MAINPROG and its subordinate load module, IOFUNC. MAINPROG contains no I/O functions, but IOFUNC has a call to **open**. Because IOFUNC is linked normally, the UNIX style I/O support routines are not linked in this load module. Because MAINPROG has no calls to **open**, UNIX style I/O support routines are not linked with MAINPROG either. To include the UNIX style I/O support routines, define **RES_UNIXIO** in the source file containing **<resident.h>** and include the generated object code when MAINPROG is linked.

If the subordinate load modules do not require preprocessing by COOL due to reentrancy or extended names considerations, COOL is not required as it is for the primary load module. Under any extended architecture system, if the primary load module is linked with AMODE=31, the subordinate load modules can be linked RMODE=ANY, that is, they can reside above the 16-megabtye line.

UNIX System Services

The use of the all-resident library is recommended for **setuid** or **setgid** shell programs. For security reasons, **setuid** and **setgid** programs ignore the ddn_CTRANS environment variable when executed. If these programs are not linked as all-resident, the SAS/C Transient Library must be in the system link list or accessible via the STEPLIB environment variable. Note that if the SAS/C Library is specified by STEPLIB, it must be present on the site-maintained list of permitted STEPLIBs for **setuid/setgid** programs.

Linking

Operating-system-dependent details on the all-resident library and the commands or control language required when linking an all-resident program can be found in Appendix 7, “Extended Names,” on page 405. For information on CICS, refer to Chapter 5, "Preprocessing, Compiling, and Linking," in the SAS/C CICS User’s Guide.

CHAPTER

11

Communication with Assembler Programs

Introduction

The compiler and library contain a number of features to facilitate the use of the C language and assembler language in the same program. The features discussed in this chapter include the following:

- the assembler macros CENTRY and CEXIT, which enable assembler modules to allocate space on the automatic storage stack
- the extensions to the C language to support call-by-reference and IBM format varying-length parameter lists so that existing assembler routines can be called easily
- the library mechanism by which a routine in another language can call a C **`main`** function, passing one or more arguments of any type.

This chapter is oriented towards applications that call assembler from C. Communicating between C and other high-level languages is discussed in detail in the SAS/C Compiler Interlanguage Communication Feature User's Guide.

Calling Conventions for C Functions

The following sections discuss calling conventions for C functions.

C Parameter Lists

Generally, C passes arguments by value rather than by reference. This means that the parameter list contains the actual argument values instead of pointers to the arguments. Suppose that the extern function **f** is called with the following arguments:

```
int i;
char c;
short s;
double d;
char *p;
f(i, c, s, d, p);
```

The parameter list generated by the compiler in this case would be mapped in assembler language as follows:

Example Code 11.1 Typical C Parameter List

```
PARMBLOK DS    0D
         DS    F         value of i
         DS    F         value of c (promoted to int)
         DS    F         value of s (promoted to int)
         DS    F         uninitialized padding bytes
         DS    D         value of d
         DS    A         value of p
```

Note that **c** has been placed in byte 3 of a word, **s** has been placed in the second halfword of a word, and **d** is aligned on a doubleword boundary. The parameter list itself is always aligned on a doubleword boundary.

Many assembler routines expect their arguments to be passed by reference instead of by value. In some cases, this type of parameter list can be generated by simply applying the ampersand (**&**) operator to each argument. However, this solution is insufficient for function calls using arguments that are not lvalues, such as constants or expressions. To support such function arguments, the compiler accepts the (nonstandard) at-sign (**@**) operator. The **@** operator can be applied only to function arguments. When applied to an lvalue, the **@** operator has the same effect as the **&** operator. When applied to a function argument that is not an lvalue (such as a constant), the **@** operator returns a pointer to a temporary copy of the value. (See “The @ operator” on page 31. Note that the **@** operator can be used only in conjunction with the **at** compiler option.)

Also, many assembler routines accept varying-length parameter lists. These routines typically expect that the last parameter has the high-order bit (the VL-bit) set to indicate that it is the final parameter. The compiler can be made to create this sort of parameter list by using the **__asm** keyword in the function declaration.

To show how the **@** operator and the **__asm** keyword can be used, suppose the function **f** in the previous example is called as follows:

```
__asm void f();
char c;
short s;
char *p;
```

```
f(@(2+3), @c, @s, @1.0, @p);
```

Then the compiler creates a parameter list, as shown in Example Code 11.2 on page 211.

Example Code 11.2 C Parameter List Using Keyword _ _asm and the @ Operator

```
PARMBLOK DS    0D
         DS    A         pointer to a temporary 5
         DS    A         pointer to c
         DS    A         pointer to s
         DS    A         pointer to a temporary 1.0
         DS    A         pointer, with VL-bit set, to p
```

An OMD listing may be extremely helpful in determining the exact format of any parameter block.

__asm, __ref, and __ibmos Keywords

The **__asm**, **__ref**, and **__ibmos** keywords are used to declare functions and pointers to functions written in assembler language that expect a parameter list in OS format.

If the **__asm** keyword is used in a declaration of a function or function pointer, the compiler creates a VL-format parameter list for the function. The compiler uses the following conditions to create a VL-format parameter list:

- If the function has no arguments, general register 1 is set to 0.
- All pointer arguments except the last argument have bit 0 set to 0. If the last argument is a pointer, bit 0 is set to 1. Bit 0 is a non-address bit, so it is safe for the compiler to modify its value.

The following declaration causes the compiler to create a VL-format parameter list for **asm_func**:

```
__asm int asm_func(void *, void *);
```

If the **__asm** keyword is used in the declaration of a function pointer, the function pointer is assumed to be local unless the **__remote** keyword is explicitly used. The following declaration causes the compiler to create a VL-format parameter list for the function called via **asm_fp**:

```
__asm int (*asm_fp)(void *, void *);
```

The **__asm** keyword does not cause the compiler to generate a call-by-reference parameter list; that is done by the **__ref** keyword, described below. You can also use the **@** (at sign) operator to pass individual parameters by reference. Refer to “The @ operator” on page 31 for more information on this operator.

The **__ref** keyword can be used in function declarations and function pointer declarations. This keyword specifies that the called function is an assembler language function expecting a call-by-reference parameter list in VL format. The effect of using the **__ref** keyword is similar to using the **__asm** keyword and the **@** operator together, with the **@** operator implied for all non-pointer arguments.

The parameter list created for functions declared with the **__ref** keyword, or called via a function pointer declared with the **__ref** keyword, contains only pointers. In general, if the argument is already a pointer type, such as **char ***, the argument is used directly without further indirection. If the argument is not a pointer type, the compiler places a pointer to the argument in the parameter list. The parameter list is in VL format as described above.

If the function is declared with a prototype, all arguments are converted to the type specified by the prototype. If the argument must be converted to match the type specified in the prototype, the compiler creates a temporary variable, assigns it the value of the argument, and passes a pointer to the temporary variable. For example, consider the declaration and call shown here:

```
__ref void myfunc(short);
int i;
i = 2;
myfunc(i);
```

The compiler creates a temporary **short** variable, assigns it the value of **i**, and places a pointer to the temporary variable in the parameter list.

Since the compiler passes a pointer to a temporary copy of the argument instead of to the argument itself, the called function cannot change the value of the argument. If changes to the argument by the called function must be reflected in the calling function, be sure to use an argument of the same type.

If the argument already has the type specified in the prototype, or if the argument is an **int** or **long** type and the prototype specifies a type that differs only in sign, no conversion is performed. For example, if the prototype specifies **unsigned int** and the argument is **signed int**, no conversion is performed and a pointer to the argument is placed in the parameter list.

Like **__asm**, if the **__ref** keyword is used in the declaration of a function pointer, the function pointer is assumed to be local unless the **__remote** keyword is explicitly used. See "Remote Function Pointers" on page 50 for more information.

Using the **__ibmos** keyword in a declaration is the same as specifying the name of the function or function pointer in a **#pragma linkage (,OS)** statement. Refer to "The #pragma linkage statement" on page 36 for more information. You may find it easier to use the **__ibmos** keyword to declare function pointers in certain situations, such as in the declaration of aggregate types. For example:

```
struct XYZ {
   /* other structure members */
   __ibmos int (*fp)(int, int);
   /* other structure members */
};
```

Function pointers declared with the **__ibmos** keyword are always local. Specification of both **__ibmos** and **__remote** results in an error.

The **__asm**, **__ref**, and **__ibmos** keywords may not be used in a declaration with any ILC function keywords, such as **__pascal**. Unless there is an attempt to convert a local function pointer to a remote function pointer, **__asm**, **__ref** , and **__ibmos** function pointers may be freely converted to each other.

Linkage Conventions

The compiler uses standard linkage when calling a function. For example, the function **f** is called with the following instructions:

```
L 15,=V(F)    R15 addresses the function.
LA 1,PARMBLOK R1 addresses the function arguments.
BALR 14,15    R14 contains the return address.
```

Also, R13 points to an 18-word save area.

The register conventions illustrated above are summarized in Table 11.1 on page 213.

Table 11.1 Register Conventions for Function Calls

General Register	Contents on Entry to the Function
1	addresses parameter list
13	18–word save area
14	return address
15	entry point address

The called function is expected to restore general registers 2 through 13 before returning. Restoring other general registers is optional. The compiler generates code when necessary to save floating-point registers before calling the function and restores them on return.

If an assembler language routine uses the access registers, and any calling routine was compiled with the **armode** compiler option, it is the responsibility of the assembler code to save the access registers on entry and restore them on exit. Note that any function called from C is called in primary address space mode.

Note: **armode** is not supported for C++ code in Release 7.00 △

Returning Values from Assembler Routines

If **f** returns a scalar value, the compiler expects the value to be in general register 15 unless the value is a **double**, **long double**, or **float**, in which case the value is expected to be in floating-point register 0. For example, suppose **f** is declared as a function returning **int**; given the call

```
val = f(i, c, s, d, p);
```

the compiler may then generate the following code sequence:

```
L     15,=V(F)
LA    1,PARMBLOK
BALR  14,15       Call f.
ST    15,VAL      Store return value in val.
```

Or, if **f** is declared as returning **double**, the compiler may generate the following:

```
L     15,=V(F)
LA    1,PARMBLOK
BALR  14,15       Call f.
STD   0,VAL       Store return value in val.
```

If function **f** returns a structure or union value, the linkage is a little more complicated. In this case, a pointer to an area in which the return value should be stored is located 4 bytes before the parameter list. This pointer may be 0 if the function result is discarded as the result of being cast to **void**. In addition to copying the return value to the area addressed by the return value pointer, the function must also clear the pointer before returning, or a later call whose return value is discarded may cause overlay of the previous return value.

This is an example of generated code for a call to a function **S** returning a structure:

```
LA    2,SRET          Address return code.
ST    2,PARMBLOK-4    Store before parameter list.
L     15,=V(S)
LA    1,PARMBLOK
```

```
BALR  14,15
```

This is an example of generated code in **s** to return a structure value:

```
L     2,DSAPARMS          Locate incoming parameter list.
        S     2,=F'4'             Back up one word.
        ICM   3,B'1111',0(2)      Test for void return.
        BZ    NOVAL
        MVC   0(slen,3),SRET      Copy return value.
        MVC   0(4,2),CRABZERO     Zero return value address.
NOVAL   DS    0H
```

When a function returning a far pointer is called, the compiler expects its value to be returned in access register 15 and general register 15. An example of compiler-generated code to call a routine returning a far pointer follows:

```
L    15,=V(F)
LA   1,PARMBLOK
BALR 14,15        Call f.
STAM 15,15,VAL    Store returned ALET in val word 1.
ST   15,VAL+4     Store returned pointer in val word 2.
```

When a function returns a **long long** (or **unsigned long long** value), the result is returned in registers 15 and 0. Register 15 contains the high-order 4 bytes, and register 0 the low-order 4 bytes.

Adding Assembler Routines to C Programs

Most existing assembler routines can be called from C with little modification, if any. Furthermore, assembler routines called from C can call other C routines if, when C is called, register 12 has the same value that it had when the first assembler routine was entered. (Additional restrictions apply if one of the run-time linkage options **=optimize** or **=minimal** is in use, as described later in this chapter.) In compiled code, general register 12 always addresses a block of data known as the C Run-Time Anchor Block (CRAB). The CRAB is explained in detail in "The C Run-Time Anchor Block" on page 215.

Adding Existing Assembler Routines to C Programs

Many existing assembler routines can be called from C without modification. Unless the routine expects a normal C parameter list, you may need to use one or more of the **@** operators or the **__asm**, **__ref**, or **__ibmos** keywords to cause the compiler to create the parameter list in the format expected by the assembler routine.

Adding New Assembler Routines to C Programs

Writing an assembler routine for use only by C programs is a relatively simple task. In general, the routine should expect a normal C parameter list and follow the register conventions described earlier. Assembler routines can call functions written in C if general register 12 addresses the CRAB when the C function is called. If the CENTRY and CEXIT macros are used, C library functions can be called.

Using Macros, Control Blocks, and DSECTs

A number of assembler macros provided on the SAS/C installation tape are useful when communicating with assembler. These macros, and the control blocks they describe, are discussed in this section.

The C Run-Time Anchor Block

The C Run-Time Anchor Block (CRAB) is the primary control block for the C library. Compiled code depends upon general register 12 addressing the CRAB. Among other things, the CRAB contains the following:

- frequently used constants
- temporary work areas
- addresses of data objects such as the pseudoregister vector
- global data for library functions
- user words.

Some of the fields in the CRAB can be conveniently used in an assembler routine. Note especially the fields in Table 11.2 on page 215.

Table 11.2 Useful CRAB Fields

Label	Hex	Decimal	Description
CRABZERO	18	24	a **double** and **int** 0
CRABDBL1	20	324	a **double** 1.0
CRAB2P31	28	40	constant for **double** <->**int** conversions
CRABUNMO	30	48	unnormalized **double** 0.0
CRABDWK	38	56	**double** <->**int** conversion work area
CRABINTI	44	68	an **int** 1
CRABNEGI	48	72	an **int** minus 1
CRABEOST	50	80	pointer to**strlen** translation table
CRAB16M	54	84	an **int** 16777215
CRABTAUT	90	144	a 112-byte work area
CRABUSR1	110	272	user word 1
CRABUSR2	114	276	user word 2
CRABUSR3	118	280	user word 3
CRABTUSR	11C	284	user word 4

Note: The distributed CRAB macro defines only those library fields associated with the systems programming environment (SPE). Some of these fields are used for other purposes when the full SAS/C Library is used. △

The uses of the CRAB constant fields such as CRABINT1 are obvious. An explanation of the use of the **double** to **int** (and **int** to **double**) conversion fields is beyond the scope of this discussion, but an examination of the generated code for such conversions (via an OMD listing) shows how the conversions are performed. Similarly, the generated code for a built-in **strlen** function call shows the use of the translation table addressed by CRABEOST. The work area at label CRABTAUT is used by the compiler for functions with small automatic storage requirements and can be used by any C function.

Cautions

Keep in mind the following concerning the CRAB:

- Since any C function can use this area, the data in CRABTAUT may not be relied upon across function calls.
- The data in the CRAB, with the exception of those areas specifically intended to be used as temporary work areas, cannot be changed. Both the compiler and the library rely on these data items. Modification of CRAB data that are intended to be constant causes unpredictable results, including incorrect computations and abends.

The CENTRY and CEXIT Assembler Macros

It is possible for assembler functions to be coded to use run-time facilities such as stack allocation and inclusion in abend tracebacks. Functions that make use of the C stack can be made reentrant more easily, and their display in an abend traceback (which prints the function name and the offset in the function) makes debugging abends easier. To make use of these facilities in the same way as compiled code, each function must begin with a CENTRY macro and return via the CEXIT macro.

In a program that runs with the **=optimize** or **=minimal** run-time linkage option, assembler routines must use the CENTRY and CEXIT macros if they call C functions.

When you use CENTRY and CEXIT, you must supply a CSECT statement before the first entry point; conventionally, the CSECT name should be the name of the first entry point, followed by the **@** operator. (You can use some other name without adverse consequences if the name is not the same as another external name in the load module.) Programs that use CENTRY and CEXIT should also issue the CREGS macro to define symbolic registers and should copy the members CRAB and DSA to obtain mappings of these C run-time control blocks. Assembler functions that use CENTRY and CEXIT should ensure that general register 12 addresses the CRAB when entering and exiting the function (unless the CENTRY INDEP=YES parameter is used).

These macros and members are included in the assembler LCUSER MACLIB (under CMS) or SASC.MACLIBA (under OS/390).

The CENTRY macro

This is the form of a call to the CENTRY macro:

```
label    CENTRY DSA=dsa-size,
                BASE=base-reg,
                FNM=function-name,
                STATIC=NO/YES,
                INDEP=NO/YES,
                LASTREG=last-reg
```

All the keyword parameters are optional. The label of the CENTRY macro is the name of the entry point. It is defined as an external symbol unless STATIC=YES is specified. The keyword parameters are described below:

DSA =*dsa-size*
specifies the size of the routine's Dynamic Save Area (DSA); if DSA is omitted, a minimum DSA (120 bytes) is allocated. In addition to providing the save area for called functions, the DSA can be used as a storage area for **auto** variables. Specify DSA=0 to avoid allocation of a DSA. DSA=0 can be used only for routines that

- call no other routines
- do not issue system macros that modify the storage area addressed by register 13.

BASE=*base-reg*
specifies a base register for the routine. If BASE is omitted, R9 is assumed.

FNM=*function-name*
specifies a function name for the assembler routine. This is the name that appears in an error traceback to identify the function. If no FNM keyword appears on the macro call, the value of *label* is assumed.

STATIC=NO|YES
determines whether the function is to be externally defined. The default is STATIC=NO.

INDEP=NO|YES
determines whether the **indep** form of function linkage is required. INDEP=NO is the default. INDEP=YES is required if the assembler routine can be called from a routine that does not preserve the C execution framework pointer normally contained in register 12. The INDEP=YES linkage is less efficient than the INDEP=NO linkage and requires that L$UPREP be linked with the routine that uses the CENTRY macro. Refer to Appendix 6, "Using the indep Option for Interlanguage Communication," on page 393 for more information.

LASTREG=*last-reg*
specifies the last register to be saved for this routine. You can specify any register between R6 and R11. If no register is specified, R11 is assumed. All registers between R14 and the LASTREG value are saved when CENTRY is executed and restored when CEXIT is executed. If any unsaved registers are modified, the effects are unpredictable. If INDEP=YES is specified, the value of LASTREG is ignored, and registers R14 through R12 are always saved.

When the CENTRY macro is expanded, a USING CRAB,R12 statement should be in effect. You can use the USING positional operand of the CREGS macro to generate such a USING statement automatically.

The CEXIT macro

The CEXIT macro returns control from a routine that begins with a call to CENTRY. The form of a CEXIT call is as follows:

```
label     CEXIT RC=return-info/(reg),
                  DSA=YES/0,
                  INDEP=NO/YES,
                  LASTREG=last-reg
```

The keyword parameters are described below:

RC=*return-info*|(*reg*)
specifies an integer constant to be returned as the value of the returning function. In this case, the value is returned in R15.

You can specify any general purpose register except for register 1 (R1). R1 is used by the CEXIT macro. Specifying R1 as the value for RC will prevent the return code from being stored correctly.

Alternately, RC=(*reg*) specifies a register containing the return value. If RC is omitted, no value is returned unless the assembler routine is declared as returning **`double`**. In this case, do not use the RC keyword; instead load the return value into floating-point register 0 before issuing the call to CEXIT.

DSA=YES|0
must equal 0 for CEXIT if the corresponding CENTRY macro specifies DSA=0. Otherwise, DSA can be omitted.

INDEP=YES
should be specified if the corresponding CENTRY macro also specifies INDEP=YES.

INDEP=NO
should be specified (or the INDEP option omitted entirely) if the corresponding CENTRY macro does not specify INDEP=YES.

LASTREG=*last-reg*
specifies the last register to be restored on return from this routine. This specification should always match the LASTREG specification on the corresponding CENTRY macro.

The CREGS Macro and the CRAB and DSA DSECTs

Another group of facilities useful for the assembler programmer is the CREGS macro and the DSECTs CRAB and Dynamic Save Area (DSA), which map C run-time control blocks.

CREGS can be issued in the form CREGS USING to obtain appropriate USING statements for the CRAB and the DSA.

The CRAB DSECT should be copied because it is required for the proper expansion of CENTRY and CEXIT.

The DSA DSECT can be copied to obtain a map of the standard part of the DSA. After the standard part of the DSA, you can define additional DSA fields and then use the EQU operator to compute a total DSA size for use in CENTRY. The additional fields can be used as automatic variables. For an example of defining **`auto`** variables in an assembler function, see Example Code 11.3 on page 218.

Example Code 11.3 Defining Auto Variables in the DSA

```
function body

         COPY  DSA
TEMPVAR  DS    F               auto int variable
SHORTX   DS    H               auto short variable
STR1     DS    CL40            auto array of char
DSALEN   EQU   *-DSA           compute total length of DSA
         END
```

If your assembler function does not define any automatic variables but does call another function, the size of the minimal DSA needed in this case is defined by the symbolic name DSAMIN.

Note that CENTRY saves a pointer to the parameter list in the DSA at label DSAPARMS (offset 80, X'50').

Calling an Assembler Routine from C

This is a simple example of a C **main** program that calls an assembler routine named **SUMINT**. This example is used as the main driver function for Example Code 11.5 on page 220 and Example Code 11.8 on page 223.

Example Code 11.4 Sample C main Program

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

/* Function Declaration for the assembler routine                    */
extern __asm int sumint(int *, ...);/* Note:                          */
                                    /*   1) __asm keyword will build */
                                    /*      a VL-format parameter list.*/
                                    /*      If the last argument is  */
                                    /*      a pointer, the high order*/
                                    /*      bit of byte 0 will be set*/
                                    /*      on.  SUMINT expects all  */
                                    /*      arguments to be pointers */
                                    /*      to int.                  */
                                    /*                               */
                                    /*   2) Usage of the ellipsis    */
                                    /*      to indicate a variable   */
                                    /*      length parameter list   */
                                    /*      should be expected.      */

/* The following demonstrates how to specify runtime options in      */
/* in source code.  They are not required for the proper execution   */
/* of the C#ASM sample.                                              */
extern int _options = _VERSION + _BTRACE + _USAGE + _WARNING;

int main ()
{
    int h = 1;
    int i = 2;
    int j = 3;
    int k = 4;
    int sum = 0;              /* Returned value                      */
    int check_sum;            /* Check variable                      */
    int retcode = 0;

    /*-------------------------------------------------------------*/
    /* First Time with 2 argument pointers to int                  */
    /*-------------------------------------------------------------*/
    sum = sumint(&h, &i);         /* Note: Args passed by reference. */

    printf("\n\nVariable Length with 2 arguments, "
```

```
            "the sum of %d and %d is %d\n", h, i, sum);

    check_sum = h+i;
    if (sum == check_sum)  /* Verify Sum is corrrect.*/
    {
        printf("\nSum of %d was correct.\n", sum);
        retcode = 0;
    }
    else
    {
        printf("\nSum of %d was NOT CORRECT!.\n", sum);
        printf("It should have been %d!\n",check_sum);
        retcode = 12;
    };

    /*-------------------------------------------------------------*/
    /* Second Time with 4 argument pointers to int                 */
    /*-------------------------------------------------------------*/
    sum = sumint(&h, &i, &j, &k); /* Note: Args passed by reference. */

    printf("\n\nVariable Length with 4 arguments, "
           "the sum of %d, %d, %d and %d is %d\n",
            h, i, j, k, sum);

    check_sum = h+i+j+k;
    if (sum == check_sum) /* Verify Sum is correct.*/
       {printf("\nSum of %d was correct.\n", sum);}
    else
    {
        printf("\nSum of %d was NOT CORRECT!\n.", sum);
        printf("It should have been %d!\n",check_sum);
        retcode = 12;
    };

    exit(retcode);
}
```

Example Code 11.5 on page 220 is a simple example of an assembler routine that returns the sum of integers to its caller. Since no functions are called from the assembler routine and no automatic storage is needed, the CENTRY and CEXIT macro parameter DSA=0 defines a function with small automatic storage requirements.

Example Code 11.5 Sample Assembler Routine Using CENTRY and CEXIT

```
         EJECT
         PRINT ON,GEN
SUMINT@  CSECT
         CREGS USING
         SPACE
SUMINT   CENTRY INDEP=NO,DSA=0
*-------------------------------------------------------------------*
* Make sure we actually got a plist address on the call.            *
*-------------------------------------------------------------------*
         SR    R3,R3             Clear R3 for sum'ing
         LTR   R1,R1             Is there a plist?
```

```
         BZ    DONE              Nope, just leave w/R3=0!
         SPACE
*-------------------------------------------------------------------*
* Sum integers passed via VL-format parameter list.                  *
*-------------------------------------------------------------------*
         SR    R3,R3             Clear R3 for summing
NEXTADD  DS    0H
         L     R4,0(R1)          Load pointer
         A     R3,0(R4)          Add integer to sum
         TM    0(R1),X'80'       End of VL-Plist? <---Note This Check
         BNZ   DONE              Yes, finish up and exit
         LA    R1,4(R1)          No, bump to next argument
         B     NEXTADD           Start again
DONE     DS    0H                Yes, prepare to return
         SPACE
*-------------------------------------------------------------------*
* Exit with the sum of the integers provided to CEXIT in R3.        *
*-------------------------------------------------------------------*
         CEXIT  RC=(R3),INDEP=NO,DSA=0
         EJECT
*-------------------------------------------------------------------*
* Constants                                                         *
*-------------------------------------------------------------------*
         LTORG                        Area for Literal Pool
*-------------------------------------------------------------------*
* Working Storage                                                   *
*-------------------------------------------------------------------*
         COPY  DSA                    Required for CENTRY/CEXIT
*-------------------------------------------------------------------*
* Dsects                                                            *
*-------------------------------------------------------------------*
         COPY  CRAB                   Required for CENTRY/CEXIT
         END   SUMINT
```

Example Code 11.6 on page 221 and Example Code 11.7 on page 222 are examples of a C **`main`** program calling an assembler routine that issues an EXEC CICS READ command. The assembler routine is passed a file key as a parameter; it then returns a pointer to the record that was read. The EXEC CICS command is translated into an invocation of the DFHECALL macro. This macro uses a work area to build a parameter list to pass to CICS. Storage for the parameter list is allocated in the DSA.

Example Code 11.6 Sample C main Program Calling a CICS Assembler Application

```
#pragma options copts(dollars)
#include <stdio.h>
void main()
{
void *readrec();
struct DFH$AFIL {
    char filea [0] ;
    char filerec [0] ;
    char stat;
    char numb [6] ;
    char name [20] ;
    char addrx [20] ;
```

```
    char phone [8] ;
    char datex [8] ;
    char amount [8] ;
    char comment [9] ;
} *dfh$afil;

dfh$afil = readrec("111111");

if (!dfh$afil) printf(" read failed\ n");

else printf(" %.20s\ ", dfh$afil->name);
}
```

Example Code 11.7 CICS Assembler Application Routine

```
*ASM XOPTS(NOPROLOG NOEPILOG)
READREC@ CSECT
         CREGS USING              Register equates.
READREC  CENTRY DSA=DSALEN        Generate C prolog.
         SPACE
         L     R3,0(,R1)          Point to passed parameter.
         SPACE
         EXEC CICS ADDRESS EIB(R4)
         SPACE
         USING DFHEIBLK,R4        Establish EIB addressability.
         SPACE
         EXEC CICS READ FILE('FILEA') SET(R15) RIDFLD(0(,R3)) RESP(RC)

         SPACE
         CLC   RC,DFHRESP(NORMAL) Check command response code.
         BE    RETURN             Branch if OK.
         L     R15,CRABZERO       Else return null.
         SPACE
RETURN   CEXIT RC=(15)            Return pointer to record or null.
         SPACE
         DROP  R4                 end of EIB addressability
         EJECT
         COPY  DSA
DFHEIPL  DS    20F                used by DFHECALL macro for parm list

DFHEITP1 DS    F                  used by DFHECALL for return info
RC       DS    F                  EXEC CICS command response code
DSALEN   EQU   *-DSA
         EJECT
         COPY  CRAB               CRAB control block map
         EJECT
DFHEIBR  EQU   0
         COPY  DFHEIBLK           EIB map
         SPACE 2
         END
```

Calling a C Function from Assembler

A function written in C can be called from an assembler function as long as general register 12 addresses the CRAB when the C function is called. If the function is a library function, the calling (assembler) function must use the CENTRY and CEXIT macros to preserve the DSA chain. Most library functions depend upon being called from a normal C framework, that is, with general register 12 addressing the CRAB and general register 13 addressing a C DSA.

Example Code 11.8 on page 223 shows the **sumint** function expanded to call **printf** to write the result to **stdout**. Note that the **printf** parameter list is created in the DSA and that the CENTRY parameter DSA now specifies a non-zero DSA size. This version of **sumint** is called from the C **main** program in Example Code 11.4 on page 219.

Example Code 11.8 Calling a C Library Function from Assembler

```
         EJECT
         PRINT ON,GEN
SUMINT@  CSECT
         CREGS USING
         SPACE
SUMINT   CENTRY INDEP=NO,DSA=DSALEN
*-------------------------------------------------------------------*
* Make sure we actually got a plist address on the call.            *
*-------------------------------------------------------------------*
         SR    R3,R3              Clear R3 for sum'ing
         LTR   1,1                Is there a plist?
         BZ    DONE               Nope, just leave w/R3=0!
         SPACE
*-------------------------------------------------------------------*
* Sum integers passed via VL-format parameter list.                   *
*-------------------------------------------------------------------*
         SR    R3,R3              Clear R3 for summing
NEXTADD  DS    0H
         L     R4,0(R1)           Load pointer
         A     R3,0(R4)           Add integer to sum
         TM    0(R1),X'80'        End of VL-Plist? <---Note This Check
         BNZ   DONE               Yes, finish up and exit
         LA    R1,4(R1)           No, bump to next argument
         B     NEXTADD            Start again
DONE     DS    0H                 Yes, prepare to return
         SPACE
*-------------------------------------------------------------------*
* Call printf to display the sum of integers                        *
*-------------------------------------------------------------------*
         ST    R3,SUMINTS         SUM of int's to Parmlist
         MVC   FMTPTR,=A(FORMAT)  Move format pointer to PRINTF
*                                 Parmlist
         L     R15,=V(PRINTF)     R15 -> PRINTF
         LA    R1,PARMLIST        R1  -> Parmlist Address
         BALR  R14,R15            Call PRINTF
         SPACE
*-------------------------------------------------------------------*
* Exit with the sum of the integers provided to CEXIT in R3.        *
```

```
*--------------------------------------------------------------------*
         CEXIT  RC=(R3),INDEP=NO
         EJECT
*--------------------------------------------------------------------*
* Constants                                                          *
*--------------------------------------------------------------------*
         LTORG                         Area for Literal Pool
FORMAT   DC    X'15'                   New Line Before Output
         DC    C'Assembler Sum: %d'
         DC    X'1500'                 New line with NULL terminator
*--------------------------------------------------------------------*
* Working Storage                                                    *
*--------------------------------------------------------------------*
         COPY  DSA                     Required for CENTRY/CEXIT
PARMLIST DS    0D
FMTPTR   DS    A                       Address of printf parmlist
SUMINTS  DS    F                       Sum of int's
DSALEN   EQU   *-DSA                   Length of DSA
*--------------------------------------------------------------------*
* Dsects                                                             *
*--------------------------------------------------------------------*
         COPY  CRAB                    Required for CENTRY/CEXIT
         END   SUMINT
```

Calling a C Program from Assembler

Before a C program can be executed, the C execution framework must be created. Normally, the framework is created by the library routine L$CMAIN, which is defined by the linkage editor to be the first routine executed in a C load module. L$CMAIN expects to be called by the operating system and therefore expects a standard OS/390 or CMS format parameter list, consisting of a character string plus various system-dependent format information. L$CMAIN processes this information, transforms it into the C standard **argc** and **argv** format, and calls the C **main** function with the constructed **argc** and **argv**. L$CMAIN can be called directly from assembler to pass control to a **main** C routine via the normal C entry point, MAIN. However, invoking a C program via MAIN is rarely convenient because the type of parameter list required is both inflexible (allowing only character data to be passed) and operating system dependent.

To avoid this problem, two additional entry points, named $MAINC and $MAINO, are provided to L$CMAIN.

Note: The behavior of $MAINC and $MAINO in CICS is different than the behavior of these entry points under OS/390 or CMS. See Example Code 11.11 on page 226. △

Entry point $MAINC expects to receive a list of addresses in the standard OS VL-type parameter list format. $MAINC transforms the input parameters into the standard C **argc** value (number of arguments plus 1) and the **argv** vector. Each element of **argv** after **argv[0]** contains the corresponding address from the input parameter list. (For example, **argv[1]** contains the first address from the list.)

Entry point $MAINO expects a list of addresses in the standard OS VL-type parameter list format. The first argument to $MAINO is a pointer to a string containing run-time options, preceded by a halfword containing the number of characters in the string. The first word in the argument list should address the prefix, not the string itself. This information is processed by the run-time library and is not

passed to the C **main** program. Each element of argv after **argv[0]** contains the corresponding address from the input parameter list. (For example, **argv[1]** contains the second address from the list, which represents the first argument.)

Example Code 11.9 on page 225 shows an assembler program that calls a C function through $MAINO. The C program using the argument as passed by assembler through $MAINO is in Example Code 11.10 on page 225.

Example Code 11.9 Calling a C main Function from Assembler via $MAINO

```
MAINASM  CSECT
         STM   14,12,12(13)            standard OS entry linkage
         BALR  9,0
         USING *,9
         LR    14,13
         LA    13,SAVEAREA
         ST    14,4(13)
         ST    13,8(14)                end of standard entry
*
*  Assembler segment that calls $MAINO
*
         LA    1,PARMLIST
         L     15,=V($MAINO)
         BALR  14,15
*
*  Other processing can go here before exiting
*
MAINXT   L     13,4(13)                standard exit linkage
         LM    14,12,12(13)
         BR    14                      end of standard exit
*
         LTORG ,
SAVEAREA DC    18F'0'
PARMLIST DS    0F
         DC    A(RNTMPRM)
         DC    A(ARGV1)
         DC    A(ARGV2)
         DC    A(X'80000000'+ARGV3)
ARGV1    DC    F'42'
ARGV2    DC    D'67.4242'
ARGV3    DC    CL4'HELP'
RNTMPRM  DC    AL2(L'RNTMOPT)
RNTMOPT  DC    C'=FILLMEM =FDUMP'
         END
```

Example Code 11.10 C main Function Called from an Assembler Driver via $MAINO

```
#include <options.h>
void main(int argc,char **argv)
{
   int i;
   double f;
   char verb[4];
```

```
   i = *(int *) argv[1];
   f = *(double *) argv[2];
   memcpy(verb,argv[3],4);
}
```

Example Code 11.11 on page 226 shows sample code that calls a C program from assembler using the entry point $MAINO from CICS. CICS command-level programs are called with a parameter list of at least two entries: the address of the EXEC interface block (EIB) and the COMMAREA address. If there is no COMMAREA, a value of x'ff000000' is passed in its place.

When you use the $MAINO or $MAINC entry points, a VL-format parameter list must be passed to the library. Make sure that the last address in the list has the high-order (VL) bit set. If you are passing parameters other than the EIB and COMMAREA addresses, you cannot specify the pseudo-null value of X'FF000000' for the COMMEAREA address. The library interprets X'FF000000' as the last parameter in the list.

Example Code 11.11 Calling a C Program from Assembler via $MAINO in CICS

```
CALLMNO  DFHEIENT CODEREG=(5),
               EIBREG=,
               DATAREG=(13)       base reg = R5, dynamic storage = R13

         SPACE
         EXEC CICS ADDRESS EIB(R4)
         SPACE
         LA    R1,PGMPARMS        Point to program parms.
         ST    R1,ARGPTR          Save in parm list.
         ST    R4,ARGV1           Save address of EIB.
         MVC   ARGV2,=X'FF000000' Indicate no commarea.
         LA    R1,PARMLIST        Point to the parm list.
         L     R15,=V($MAINO)     Call the C program.
         BALR  R14,R15
         SPACE
         DFHEIRET
         SPACE
PGMPARMS DS   0H
         DC   AL2(L'ARGSTR)       length of run-time argument string
ARGSTR   DC   C'=56K =storage'
         SPACE
         DFHEISTG
         SPACE
PARMLIST DS   0F
ARGPTR   DS   A                 pointer to run-time arguments
ARGV1    DS   A                 argv [1]  pointer to the EIB
ARGV2    DS   A                 argv [2]  pointer to any commarea
         SPACE
         DFHEIEND
         SPACE
         CREGS
         SPACE
         END
```

CHAPTER

12

Simple Interlanguage Communication

Introduction

This chapter explains how to call a main program in one language from a main program in another. Additional interlanguage communication information can be found in the SAS/C Compiler Interlanguage Communication Feature User's Guide. This publication describes how to write more complicated multilanguage applications, such as a COBOL main program that calls C subroutines.

Appendix 6, "Using the indep Option for Interlanguage Communication," on page 393 contains information on C functions calling or being called by other languages without the use of the interlanguage communication (ILC) feature. Use of the ILC feature is highly recommended for new programs. Appendix 6, "Using the indep Option for Interlanguage Communication," on page 393 explains the techniques that were required before this feature was available.

An Overview of Interlanguage Communication

Communication between C and another high-level language such as FORTRAN or PL/I follows the same principles as communication between C and assembler language. (See Chapter 11, "Communication with Assembler Programs," on page 209 for more information on communicating with assembler language programs.) However, a high-level language introduces several potential complications into the communication.

First, many high-level languages require their own execution framework, or environment. (Execution frameworks are also discussed in more detail in Appendix 6, "Using the indep Option for Interlanguage Communication," on page 393.) The other language's framework must be active when a routine in that language receives control. This is in contrast to assembler language programs, which can execute with the C execution framework still active. When control passes to C, the C framework must be active. Therefore, the appropriate framework must be activated whenever control passes across a language boundary.

Second, a high-level language may not be able to create parameter blocks in the format expected by a C program, and C may not be able to create parameter blocks in the format expected by another language.

Third, the other high-level language may not support every C data type or may support additional data types with no corresponding C data types.

Last, error handling can be complex. It is sometimes important to ensure that the language in which an error occurs is the one that handles the error.

Calling a C main Function from Another Language

Calling a C **main** function from another high-level language is the easiest case of interlanguage communication. Provided that the other language's compiler produces a call-by-reference parameter list, as the IBM FORTRAN, PL/I, and COBOL Compilers do, you can simply invoke $MAINC or $MAINO, as appropriate, from the other language, as described in Chapter 11, "Communication with Assembler Programs," on page 209. Calling one of these entry points initializes the C execution framework, and that framework is accessible when the C program assumes control. Example Code 12.1 on page 228 shows a FORTRAN call to a C **main** function. (The function called is the same function shown in Example Code 11.10 on page 225. The FORTRAN call has exactly the same effect as the assembler call included in that example.)

Example Code 12.1 A FORTRAN Call to a C main Function (via $MAINO)

```
      CHARACTER*17 COPTS
      INTEGER*2 OPTLEN
      EQUIVALENCE (COPTS, OPTLEN)
C
C  CALL A MAIN C ROUTINE, PASSING THE RUN-TIME OPTIONS =FILLMEM AND
C  =FDUMP.  THE OPTIONS STRING MUST BE PRECEDED BY A HALFWORD
C  CONTAINING THE STRING LENGTH.
C
      COPTS = 'NN=FILLMEM =FDUMP'
      OPTLEN = 15
      CALL $MAINO(COPTS, 42, 67.4242D0, 'HELP')
```

Your C **main** function can, in turn, call other C functions or assembler language routines. However, **main** cannot call subroutines written in the other language. If you need to do this, refer to the SAS/C Compiler Interlanguage Communication Feature User's Guide or Appendix 6, "Using the indep Option for Interlanguage Communication," on page 393.

Your C **main** function should return to the other language when it has finished by executing a **return** statement or by calling **exit**. Either terminates the C execution framework and returns control to the other language. The exit or return value is passed back in register 15, where it can be accessed if the other language provides this capability (as do COBOL and PL/I). Note that you can use **exit** in any C function to return to the calling program.

A C execution framework is created and destroyed (on return) every time you call a C **main** function. This could possibly create a significant overhead if the function is called many times. If this overhead is a problem in your application, you should consider using the interlanguage communication (ILC) feature.

Note that you can call only one **main** C function per load module in this manner because of the need to route all calls through one of the $MAINC or $MAINO entry points. However, you can call multiple C functions from **main**, passing **main** a code so that it knows which function to call.

Calling a MAIN Routine in Another Language from C

Calling a MAIN routine in another language from a C program is also fairly straightforward. Call the other language at the entry point described in the documentation for the other language. For example, PL/I is called at PLISTART, PLICALLA, or PLICALLB. You must build a parameter list or parameter block in the format expected by the implementation of the other language and pass its address in the manner expected by the other language. In most cases, you use the standard OS/390 parameter list format. For some languages, such as PL/I, you may be able to handle parameter passing with C code; for others, you may need to write an assembler stub to do it. Use of the `@` operator or the **`__ref`** keyword can assist you in building a call-by-reference parameter list. See Chapter 11, "Communication with Assembler Programs," on page 209 for more information on these features.

Because you are calling a MAIN routine, the other language's execution framework is set up before the other language program receives control, and it is terminated when the other language routine ends. You do not need to be concerned with the details of the other language's framework. You can call subroutines in the other language from the other language's MAIN routine, and you also can call assembler subroutines (subject to the restrictions imposed by the other language). However, you cannot call C functions from other languages. If you need to do this, use the ILC feature or the techniques described in Appendix 6, "Using the indep Option for Interlanguage Communication," on page 393.

CHAPTER

13

Inline Machine Code Interface

Introduction

The SAS/C inline machine code interface feature extends the capabilities of your C program by enabling you to write more efficient code and to incorporate instructions that cannot normally be generated with a high-level language. None of the facilities provided with the inline machine code interface are portable.

The inline machine code interface enables a C program to generate OS/390 and CMS supervisor calls (SVCs), DIAGNOSE instructions and CMS SVC 202s, and miscellaneous assembler language instructions. You can use C variables and expressions as operands for these instructions and store results from them in C variables or storage locations addressed by C pointers.

Overview

The inline machine code interface consists of the following:

- a series of built-in functions that can be used to generate OS/390 and CMS assembler language instructions. Note that although your use of these facilities has the appearance of C function calls, the built-in functions generate only inline code. No library function is called by any of the built-in functions.
- a series of macros (formatted like assembler language instructions) that can be used to issue many common assembler language instructions. In the following example, an assembler language instruction is issued by a C program by using the CS macro:

```
CS 14,15,0(1)     instruction


CS(14,15,0+b(1)); cs macro
```

  Refer to “Macros and Header Files” on page 259 for a detailed description of this example.
- three header files (**<code.h>**, **<regs.h>**, and **<svc.h>**) that provide symbolic definitions used by the built-in functions and macros.

This discussion of the inline machine code interface is intended primarily for experienced OS/390 and CMS systems programmers. It is assumed that you are already familiar with the concepts and methodologies involved in using built-in functions, supervisor calls, and assembler language interfaces.

This chapter provides the following:

- a brief overview of the inline machine code interface
- a detailed description of each built-in function
- a discussion of the macros (formatted like assembler instructions) and a listing of representative portions of the files containing the macros
- a brief discussion and partial listing of the general header files needed when you use the inline machine code interface
- an example of how to use the inline machine code interface.

Built-in functions

The SAS/C Compiler provides the following set of built-in functions that enable you to use machine code in C programming applications:

_ldregs loads register values.

_stregs	stores register values.
_cc	tests or saves the hardware condition code.
_diag	generates a DIAGNOSE instruction.
_cms202	generates a CMS SVC 202 instruction.
svc202 or **e_svc202**	generates a CMS SVC 202 instruction with arguments.
_ossvc	generates an OS/390 SVC instruction.
_osarmsvc	generates an OS/390 SVC instruction to be issued in AR mode
_code	generates a machine instruction or inline data.
_label	defines an inline-code branch target.
_bbwd	branch backward to a previously defined label.
_bfwd	branch forward to a label defined later.
_branch	alternate form of **_bbwd**/**_bfwd** for use in library code macros.

Each of the built-in functions is discussed in "Functions" on page 239.

_ldregs, _stregs, and _cc

Two framing functions, **_ldregs** and **_stregs**, enable you to set register values before issuing an instruction or series of instructions and then retrieve values from the registers after the instruction sequence is completed. Between these framing functions, you can call other built-in functions or issue macros that resemble assembly instructions. You can use the **_cc** function to access the condition code set by a generated DIAGNOSE, SVC, or machine instruction.

_diag, _cms202, and _ossvc

Several built-in functions issue specific supervisor call instructions: **_diag**, **_cms202**, and **_ossvc**. Access to the DIAGNOSE instruction through the **_diag** function opens the way for using the multitude of CMS diagnose codes. With the **_diag** function, your program can interact with the console, examine real storage, read the system symbol table, execute timing functions, and so on. The **_ossvc** and **_osarmsvc** built-in functions can be used to invoke an OS/390 or CMS supervisor call to get access to many supervisor services that would otherwise be available only through the Assembler language. The **_ossvc** built-in function always issues the SVC in primary address space mode, while the **_osarmsvc** issues the SVC in access register mode if the compiler option has been specified.

_code

The inline machine code interface also provides a very flexible method for generating machine instructions or inline data. The **_code** function enables you to generate machine instructions directly from C without the overhead of a subroutine call.

Furthermore, the library provides header files defining macros to assist you in using **_code** to generate machine instructions.

_label

The **_label** function defines a location in an inline code block and associates it with a non-zero **unsigned short** constant. You can transfer control directly to a label defined with **_label** by using the **_bbwd**, **_bfwd**, or **_branch** function, or indirectly by using a library macro which calls one of these functions. The same label value may be assigned more than once in a compilation. The **_bbwd** and **_bfwd** functions always branch to the nearest definition of the target label in the appropriate direction. This property allows **_label** to be used in macros that are expanded repeatedly in a single compilation.

_bbwd and _bfwd

The **_bbwd** and **_bfwd** functions allow you to generate selected 370 branch instructions to labels defined with **_label**. The macro specifies the particular branch operation and any non-target information required by the instruction, for example, count registers.

All use of these functions and macros must be localized within a single block of inline code, beginning with an **_ldregs** call, and containing no C statements other than calls to inline machine code functions. The results of attempting to branch from one code block to another are unpredictable.

_branch, _blabel, and _flabel

The **_bbwd** and **_bfwd** functions are not convenient for symbolic use. For this reason, the **_branch** function and the **_blabel** and **_flabel** macros are provided. These are more complicated than **_bbwd** and **_bfwd**, but lend themselves more readily to symbolic use. **_branch** is a variant of **_code** in which the final halfword of an instruction can be specified in one of two special forms larger than a halfword. One of these forms is generated by **_blabel**, and the other by **_flabel**. To illustrate, the **BCT** macro defined in the **<genl370.h>** header file (after expansion of various inner macros) has the form:

```
#define BCT(r1,x,s)  _branch(0x80000000 >> (r1),
                             0x4600 | ((r1) << 4) | (x), s)
```

The following uses of the BCT macro then behave as follows:

```
BCT(R1, 0, 6+b(3))      /* generates a BCT   1,6(0,3)      */
BCT(R1, 0, _blabel(2))  /* same effect as _bbwd(0x4610, 2) */
BCT(R1, 0, _flabel(2))  /* same effect as _bfwd(0x4610, 2) */
```

The macros in **<genl370.h>** for instructions supported by **_bfwd** and **_bbwd** have all been defined to use **_branch**.

Code Macros

Although the **_code** function provides a very flexible method for generating instructions, many machine instructions can be more easily generated using code macros. The macros available for use are provided in the following series of header files:

- **<genl370.h>**
- **<str370.h>**
- **<dec370.h>**
- **<float370.h>**

- **<ctl370.h>**
- **<supv370.h>**
- **<das370.h>**
- **<io370.h>**
- **<ioxa.h>**
- **<vec370.h>**
- **<lsa320.h>**

These header files provide appropriate macros for all IBM 370 machine instructions except UPT, SIE, and PC. (These three instructions cannot be supported because of conflicts in register use between the instructions and the compiled C code.)

Two other header files, **<code.h>** and **<regs.h>**, define basic-level macros that are used by the macros in the header files listed above. You can use the macros in **<code.h>** to simplify the arguments to the **_code** function. All of these header files are described in detail in "Macros and Header Files" on page 259.

Bit Masks for Using Registers

Several of the built-in functions use a 32-bit **mask** argument to indicate which registers the generated machine instructions should use. Starting from the left of the mask, bits 0 through 15 indicate whether general purpose registers 0 through 15 are used. Bits 16, 18, 20, and 22 indicate the use of floating-point registers 0, 2, 4, and 6, respectively. The remaining bits are not currently used and should be specified as 0.

The **<regs.h>** header file, which is included (via **#include**) in both **<svc.h>** and **<code.h>**, contains macros named R0 through R15 for general registers 0 through 15, and F0, F2, F4, and F6 for floating-point registers 0, 2, 4, and 6. These macros enable you to symbolically specify the register mask. For example, coding the **mask** argument in the following way sets the bit mask to 0xc0010000, which requests the use of registers 0, 1, and 15:

```
R0+R1+R15
```

Table 13.1 on page 238 summarizes the use of registers by **_code**, **_ldregs**, and **_stregs**.

When you compile with the **armode** option, you can use the inline machine code functions to execute instructions that modify the access registers (for instance, CPYA). An access register is considered to have the same register number as the corresponding general register. For example, the bit mask R2 indicates that general register 2, access register 2, or both may be modified by the instruction. See Chapter 15, "Developing Applications for Use with UNIX System Services OS/390," on page 331 for more information on register masks.

Usage Notes

Some arguments to the built-in functions must be compile-time constants because the compiler has to know which registers or values to work with as it compiles the program.

CAUTION:

The compiler must be able to determine where inline machine code sequences begin and end. During sequences of inline machine code, you control the contents of the designated registers. In normal C code, the compiler controls the contents of all registers. Because of this difference, it is essential that the compiler be able to distinguish where inline machine code sequences begin and end. △

The following rules enable the compiler to differentiate between inline machine code and normal C code:

- An inline machine code sequence must begin with a call to **_ldregs**. If the code sequence does not require any preloaded registers, begin the sequence with **_ldregs(0)**, which informs the compiler of the start of a sequence without loading any registers. Note that **_ldregs(0)** is not the same as **_ldregs(R0)**.
- An inline machine code sequence is ended by any C code other than a call to a machine code function, including a control structure such as the **?:** operator or an **if** statement. After the occurrence of a non-machine-code construct, the contents of registers are unpredictable and, in general, will not be preserved from any previous machine code function calls.
- If you are using a call to **_stregs** or **_cc**, you must code it last in the sequence. If you use them both, code **_stregs** before **_cc**. The function **_stregs** will not change the condition code, but the function **_cc** may change register contents.
- Do not specify complex expressions as arguments to **_stregs**. The arguments should be pointer variables or the addresses of **auto** variables. If complex expressions are used, the compiler may be forced to modify values stored in registers in order to evaluate the expressions.

After the sequence ends, the compiler may again use all the registers and may generate instructions that would change the condition codes. Be careful that subsequent sequences do not depend on register values or condition code settings established in a prior sequence, because these may no longer be retained.

CAUTION:

Do not use general-purpose registers 4 through 13 with the _ldregs function. The compiler assigns general-purpose registers 6 through 11 and floating-point registers 4 and 6 to register variables. If your use of registers in **_ldregs** conflicts with the compiler's assignment of registers to register variables, the generated code may be suboptimal. General-purpose registers 0 through 3 and 14 through 15 may be used freely, as well as floating-point registers 0 and 2. △

Note that if you need to use floating-point register 4 or 6 and you specify **optimize**, you must use the **freg(0)** option to inhibit assignment of floating-point register variables.

Table 13.1 Registers for Use with the _code_stregs, and _idregs Functions

Type of Register	Register Number	Name of Macro	Bit in Mask
general-purpose	0–15	R0–15	0–15
floating-point	0	F0	16
	2	F2	18
	4	F4	20
	6	F6	22

Table 13.1 on page 238 shows the general-purpose and floating-point registers when they are used with the **_code,_stregs,** and **_ldregs** functions. No other registers are used with these functions.

The **_ldregs** and **_stregs** functions cannot be used to load or store **long long** values because a **long long** value requires two registers. You can, however, use the **_ldregs** function to put the address of a **long long** value into a register, and then use ordinary machine instructions to load or store portions of the **long long** value.

Inline Machine Code Usage Notes

Inline machine code sequences execute in access register mode if the compiler option **armode** is specified, and in primary address space mode otherwise. You can change mode within an inline code sequence so long as the mode is restored before returning to normal C code.

Note: You should be very cautious about using any access register other than access register 1 in a function not compiled with the **armode** option. Functions not compiled with the **armode** option never save or restore the access registers. As a result, a modification of an access register by a non-**armode** function can cause a calling **armode** function to behave incorrectly. △

The **_stregs** function is defined to return **int**. This allows the **_stregs** function to be used to store an integer or a pointer from a single register. Two alternate forms of **_stregs** are available to store other types of data. The function **_stfregs** returns **double**, and the function **_stpregs** returns **__far void ***. Except for the difference in return type, these functions behave the same as **_stregs**. Note that the **_stfregs** function stores its return value in the highest floating register in the mask.

Functions

The following pages define the functions for the inline machine code interface.

_bbwd

Branch to a Previously Defined Label

SYNOPSIS

```
#include <code.h>

void _bbwd(unsigned short op, unsigned short target);
```

DESCRIPTION

_bbwd causes the compiler to generate a branch instruction whose target is the previously defined label whose **_label** number is specified by **target**. The **op** argument specifies the first halfword of the instruction to generate. Both arguments to **_bbwd** must be compile-time constants.

The **op** argument must specify one of the following instructions:

BAL	**BCT**
BAS	**BXH**
BC(except **NOP**	**BXLE**

The **NOP** instruction (0x4700) is not supported. Also note that any index register in the instruction must be specified as 0.

Optimizations, such as branch folding, may cause the instructions generated as a result of **_bbwd** to differ from those expected. However, any such optimizations will not change the effects of the instructions.

CAUTIONS

If incorrect arguments are passed to **_bbwd**, the compiler produces a diagnostic, sets the return code to 8, and generates an EX 0,* instruction. This instruction causes an execute exception (OS ABEND code 0C3) if it is actually executed.

Do not use **_bbwd** to generate instructions that modify registers required by compiled code. See "_code" on page 247 for a listing of these registers.

The **_bbwd** function should only be used to transfer control to a label defined in the same block of machine code as the branch. If this rule is disregarded, the results are unpredictable.

PORTABILITY

_bbwd is not portable.

EXAMPLE

```
#include <code.h>

   /* this code multiplies the integers i and j by */
   /* repeated addition                            */

_ldregs(R1+R2+R3, i, j, 0);
_label(1);           /* LABEL1 EQU * */
_code(R3,0x1a31);    /* AR  3,1      */
_bbwd(0x4620,1);     /* BCT 2,LABEL1 */
```

RELATED FUNCTIONS

_code, **_ldregs**, **_bfwd**, **_branch**, **_label**

_bfwd

Branch to a Label Defined Later

SYNOPSIS

```
#include <code.h>

void _bfwd(unsigned short op, unsigned short target)
```

DESCRIPTION

_bfwd causes the compiler to generate a branch instruction with a target that is a label defined later. The **target** argument is the **_label** number of the target label. The **op** argument specifies the first halfword of the instruction to generate. Both arguments to **_bfwd** must be compile-time constants.

The **op** argument must specify one of the following instructions:

BAL	**BCT**
BAS	**BXH**
BC (except **NOP**	**BXLE**

The **NOP** instruction (0x4700) is not supported. Also note that any index register in the instruction must be specified as 0.

Optimizations, such as branch folding, may cause the instructions generated as a result of **_bfwd** to differ from those expected. However, any such optimizations will not change the effects of the instructions.

CAUTIONS

If incorrect arguments are passed to **_bfwd**, the compiler produces a diagnostic, sets the return code to 8, and generates an EX 0,* instruction. This instruction causes an execute exception (OS ABEND code 0C3) if it is actually executed.

Do not use **_bfwd** to generate instructions that modify registers required by compiled code. See "_code" on page 247 for a listing of these registers.

The **_bfwd** function should only be used to transfer control to a label defined in the same block of machine code as the branch. If this rule is disregarded, the results are unpredictable.

PORTABILITY

_bfwd is not portable.

EXAMPLE

```
#include <code.h>

   /* this code multiplies the integers i and j by */
   /* repeated addition.  unlike the _bbwd example,*/
   /* it behaves correctly when j is less than or  */
   /* equal to 0                                   */

_ldregs(R1+R2+R3, i, j, 0);
_code(0, 0x1222);    /*  LTR 2,2      */
_bfwd(0x4780, 2);    /*  BZ  LABEL2   */
_bfwd(0x4720, 1);    /*  BP  LABEL1   */
_code(R2, 0x1322);   /*  LCR 2,2      */
_code(R1, 0x1311);   /*  LCR 1,1      */
_label(1);           /* LABEL1 EQU *  */
_code(R3, 0x1a31);   /*  AR  3,1      */
_bbwd(0x4620, 1);    /*  BCT 2,LABEL1 */
_label(2);           /* LABEL2 EQU *  */
```

RELATED FUNCTIONS

_code, **_ldregs**, **_bbwd**, **_branch**, **_label**

_blabel

Reference a Backward Branch Target

SYNOPSIS

```
#include <code.h>

unsigned _blabel(int n);
```

DESCRIPTION

_blabel identifies a branch target occurring earlier in the compilation than the point at which **_blabel** is called. The argument **n** specifies the **_label** number of the branch target and must be a compile-time constant. A call to **_blabel** should be used only as an argument to the **_branch** function or to a macro that generates a call to this function.

RETURN VALUE

_blabel returns an encoded form of the target label that can be interpreted by the **_branch** function.

PORTABILITY

_blabel is not portable.

EXAMPLE

```
#include <code.h>
#include <genl370.h>

   /* this code multiplies the integers i and j by */
   /* repeated addition.  this code has the same   */
   /* effect as the _bbwd example, but uses macros */
   /* from genl370.h for readability               */

_ldregs(R1+R2+R3, i, j, 0);
_label(1);           /* LABEL1 EQU * */
AR(3,1);
BCT(2,0,_blabel(1))  /* BCT 2,LABEL1 */
```

RELATED FUNCTIONS

_bbwd, **_branch**, **_flabel**

_branch

Generate a Branch Instruction

SYNOPSIS

```
#include <code.h>

void _branch(unsigned mask, unsigned short op,
             unsigned int target);
```

DESCRIPTION

_branch provides a flexible method for generating branch instructions. All operands of **_branch** must be compile-time constants. Depending on the form of the **target** argument, the **_branch** function behaves as follows:

- If the target is a valid **unsigned short** value, then **_branch(mask, op, target)** has the same effect as **_code(mask, op, target)**.
- If the target has the form **0x00BBzzzz**, where **zzzz** represents any hex digits, then **_branch(mask, op, target)** has the same effect as **_bbwd(op, 0xzzzz)**. (A constant of this form is generated by the **_blabel** macro.)
- If the target has the form **0x00BFzzzz**, where **zzzz** represents any hex digits, then **_branch(mask, op, target)** has the same effect as **_bfwd(op, 0xzzzz)**. (A constant of this form is generated by the **_flabel** macro.)

CAUTIONS

_branch is not intended for direct programmer use. It is implemented primarily as a tool for use by the macros in the SAS/C inline code header files.

When a **0x00BBzzzz** or **0x00BFzzzz** argument is passed to **_branch**, all restrictions applying to the use of **_bbwd** and **_bfwd** apply to**_branch**. In particular, only permitted branch instructions can be specified, and use of an index register is not allowed.

If incorrect arguments are passed to **_branch**, the compiler produces a diagnostic, sets the return code to 8, and generates an EX 0,* instruction. This instruction causes an execute exception (OS ABEND code 0C3) if it is actually executed.

The mask operand of **_branch** defines the registers that may be modified by the generated instruction, as with the **_code** function. If the mask is not specified correctly, the effects of **_branch** are unpredictable.

Do not use **_branch** to generate instructions that modify registers required by compiled code. See "_code" on page 247 for a listing of these registers.

The **_branch** function should only be used to transfer control to a label defined in the same block of machine code as the branch. If this rule is disregarded, the results are unpredictable.

PORTABILITY

_branch is not portable.

RELATED FUNCTIONS

_bbwd, **_bfwd**, **_blabel**, **_code**, **_flabel**

_cc

Access Hardware Condition Code

SYNOPSIS

```
#include <svc.h>

int _cc(void);
```

DESCRIPTION

_cc enables you to access the condition code set by an SVC, DIAGNOSE, or machine instruction. The following table shows the hardware condition codes and their most common meanings, the **_cc** function codes, and the mnemonic macros that can be used instead of the actual values.

Table 13.2

Hardware Condition Code	Most Common Meaning of Code	**_cc** Function Returns	Symbolic Macros for **_cc** Function Return Codes
0	equal	0	CC0, CCZ, CCE
1	less	- 1	CC1, CCL, CCM
2	greater	2	CC2, CCH, CCP
3	overflow	3	CC3, CCO

.begin.end

Note: Making the **_cc** condition code {minussym}1 for the less condition (rather than 1) allows the result of **_cc** to be compared with 0 in a natural way. △

The condition code is not modified by the execution of **_cc**.

Declarations for **_cc** and the associated macros can be obtained by including (via **#include**) either **<code.h>** or **<svc.h>**.

RETURN VALUE

_cc returns the current hardware condition code as one of the integers 0, {minussym}1, 2, or 3.

CAUTIONS

If, after an SVC, DIAGNOSE, or machine instruction, you need to store registers and also access the condition code, you should use **_stregs** to store the registers first. When it is followed by a call to **_cc**, **_stregs** does not modify the condition code, but in some cases, **_cc** may modify register contents in order to address the area where the condition code is to be stored.

PORTABILITY

_cc is not portable.

IMPLEMENTATION

In general, five instructions are required to reduce the hardware condition code to an integer value. However, if the only use of the value returned by **_cc** is in a comparison with 0, only a single BC instruction is generated.

EXAMPLE

```
#include <code.h>
#include <genl370.h>
#include <stdio.h>

   /* Add two binary integers together, and print a message */
   /* if the sum is negative.                               */
int sum;
short increment;                    /* halfword */

_ldregs(R2+R3, sum, &increment);

AH(2,0,0+b(3));                     /* AH 2,0(0,3) */

if (_cc() <0)
   printf("The sum is negative\n");
```

RELATED FUNCTIONS

_cms202, **_code**, **_diag**, **_ldregs**, **_ossvc**, **_stregs**

_cms202

Generate CMS SVC 202 Instruction

SYNOPSIS

```
#include <svc.h>

void _cms202(void);
```

DESCRIPTION

_cms202 generates an SVC 202 instruction. It is intended to be used by programs executing under nonbimodal CMS. **_cms202** takes no arguments. (See **_ossvc** for generating OS/390 supervisor calls; also see the descriptions of **SVC202** and **e_SVC202** for other methods of generating an SVC 202 instruction.)

Note: The **_ossvc** built-in function may be used to generate an SVC 204 in bimodal CMS. △

RETURN VALUE

No value is returned by **_cms202**. To access values returned in registers, including any return code, use the **_stregs** function.

CAUTIONS

Use the **_ldregs** function to set up registers correctly before issuing the SVC 202.

If your program executes under OS/390, you should use the **_ossvc** built-in function rather than **_cms202**.

PORTABILITY

_cms202 is not portable.

IMPLEMENTATION

_cms202 first stores any values currently in use from general-purpose registers 0, 1, and 15 and clears any information the compiler had about the contents of those registers. It assumes that these registers are, or may be, altered by the SVC 202. **_cms202** then saves the address of the SVC instruction as an aid to traceback production in case the SVC 202 causes an abend. It then issues an SVC 202 instruction followed by a fullword (unaligned) 1.

EXAMPLE

Note: Refer to the CMS Command Reference for more details on nucleus extensions. △

```
#include <svc.h>
#include <lcio.h>
#include <stdio.h>
#include <lcstring.h>

   /* Use the CMS function NUCEXT to determine if the         */
   /* GLOBALV command is a nucleus extension.                 */
struct {                  /* NUCEXT QUERY parameter list.     */
   char cmd[8] ;          /* 'NUCEXT'                         */
   char name[8] ;         /* Nucleus extension name.          */
   char *scblock;         /* Receives pointer to SCBLOCK.     */
   char *query;           /* 0xffffffff  (identify QUERY)     */
   }nucx;


int rc;

   /* Copy in the name of the command, padded to */
   /* eight characters.                          */
memcpyp(nucx.cmd,"NUCEXT",8,6,' ');

   /* Copy in the name of the nucleus extension to be queried, */
   /* again padded to eight characters.                        */
memcpyp(nucx.name,"GLOBALV",8,7,' ');

nucx.query = (char *) -1;       /* Identify the QUERY function.*/

_ldregs(R1,&nucx);              /* R1 -> struct nucx.          */
_cms202();                      /* Issue SVC 202.              */
rc = _stregs(R15);              /* If rc == 0, GLOBALV is a    */
                                /* nucleus extension.          */

printf("Globalv %s a nucleus extension\n",
       rc == 0 ? "is" : "is not");
```

RELATED FUNCTIONS

_diag, **_ldregs**, **_ossvc**, **_stregs**, **SVC202**, **e_SVC202**

_code

Generate a Machine Instruction or Inline Data

SYNOPSIS

```
#include <code.h>

void _code(unsigned mask, unsigned short data1, unsigned short data2,
           unsigned short data3, unsigned short data4);
```

DESCRIPTION

_code provides a flexible method for generating machine instructions or inline data. **_code** enables you to issue machine instructions directly from C without the overhead of a subroutine call. In many cases, you can use either **_code** or the code macros to generate machine instructions. In general, the code macros are easier to use, but the **_code** function is more flexible. **_code** generates one to four halfwords of data into the compiled code instruction stream. Usually this is a machine instruction, but any data can be specified. **_code** takes two to five arguments that must all be compile-time constants. The first argument, **mask**, is a 32-bit mask. This argument must be a compile-time constant because the compiler has to know which registers are needed. General-purpose registers can be specified using the macros R0 through R15, and floating-point registers 0, 2, 4, and 6 can be specified as F0 through F6, respectively. Multiple registers can be specified by adding (or logically ORing) the macros, for example, R0+R1 (or R0|R1). Refer to "Bit Masks for Using Registers" on page 237 for more information on the **mask** argument.

After the register mask, the remaining arguments of **_code** are generated on a halfword boundary in the instruction stream. They are not validated in any way. Any bits in an argument that do not fit in a **short** are ignored.

The inline machine code interface provides macros for naming registers that make the register arguments to **_code** more readable when used to generate machine instructions. Refer to the listing of the **<regs.h>** header file in "General Header Files" on page 267 for the macros that can be used with **_code**.

Use the **_ldregs** function to load registers used by **_code**; use the **_stregs** function to store instruction results, and use the **_cc** function to access the condition code.

RETURN VALUE

_code returns no value. To access values returned in registers, use the **_stregs** function; to access the condition code, use the **_cc** function.

CAUTIONS

If you specify incorrect arguments for **_code** (for example, the wrong number of arguments or an invalid register mask), pass two of the compiler produces an error message, sets the return code to 8, and generates an EX 0,* instruction for **_code**. The generated instruction causes an execute exception (OS ABEND code 0C3) if it is actually executed.

Do not use **_code** to generate instructions that modify registers required by compiled code, specifically the following:

4	is used to address constants.
5	is the program base register.
12	accesses the C Run-Time Anchor Block (CRAB).
13	addresses the current automatic storage area.

If you modify any of these registers, the results are unpredictable but are likely to include an abend. See the register usage warning in "Usage Notes" on page 237.

PORTABILITY

_code is not portable.

IMPLEMENTATION

_code first stores any values currently in use from general-purpose registers specified by the register mask and then generates the requested instructions or data.

EXAMPLE

```
#include <code.h>

   /* Generate a call to the CMS DMSFREE macro, */
   /* consisting of an SVC 203 instruction,     */
   /* followed by a halfword of data.           */
unsigned amt;
char *storage;

_ldregs(R0, amt/8);

_code(R1, 0x0acb,         /* Issue SVC 203                */
          0x1e04);        /* specifying DMSFREE function. */

   /* store address of allocated memory */
storage =(char *) _stregs(R1);
```

RELATED FUNCTIONS

_cc, **_ldregs**, **_stregs**

_diag

Generate DIAGNOSE Instruction

SYNOPSIS

```
#include <svc.h>

void _diag(int n);
```

DESCRIPTION

_diag generates a DIAGNOSE instruction, using 0 and 14 as the register arguments and **n** as the diagnose code. **n** must be a compile-time constant. It is intended to be used by programs executing under CMS or some other operating system running in a virtual machine under VM/SP, VM/XA, or VM/ESA.

RETURN VALUE

_diag returns no value. To access values returned in registers, including any return code, use the **_stregs** function. To access the condition code set by **_diag**, use the **_cc** function.

CAUTIONS

Use the **_ldregs** function to set up registers correctly before issuing the DIAGNOSE. Do not use **_diag** in a real machine or when in virtual problem state.

PORTABILITY

_diag is not portable.

IMPLEMENTATION

_diag first stores any values currently in use from general-purpose registers 0, 1, 14, and 15 and clears any information the compiler has about the contents of these registers. It assumes these registers are, or may be, altered by the DIAGNOSE. It then issues a **DIAG 0**, **14**, **n** instruction, where **n** is the specified diagnose code.

EXAMPLE

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

   /* Issue DIAGNOSE X'24' to obtain information about the */
   /* virtual console.                                     */
unsigned devaddr, vinfo, rinfo;

_ldregs(R0, -1);  /* Load -1 to get virtual console information. */

_diag(0x24);      /* Ask CP for location and information.        */


   /* store the results */
_stregs(R0+R14+R15, &devaddr, &vinfo, &rinfo);

switch(_cc()) {  /* check condition code */
   case CC0:
      printf("Console address is %x\n",
            (unsigned short) devaddr);
      break;
   case CC2:
      printf("Virtual console at %x, no real console\n",
            (unsigned short) devaddr);
```

```
      break;
   case CC3:
      printf("Virtual console does not exist\n");
      break;
}
```

RELATED FUNCTIONS

_cc, **_cms202**, **_ldregs**, **_ossvc**, **_stregs**

_flabel

Reference a Forward Branch Target

SYNOPSIS

```
#include <code.h>

unsigned _flabel(int n);
```

DESCRIPTION

_flabel identifies a branch target occurring later in the compilation than the point at which **_flabel** is called. The argument **n** specifies the **_label** number of the branch target and must be a compile-time constant. A call to **_flabel** should be used only as an argument to the **_branch** function, or to a macro that generates a call to this function.

RETURN VALUE

_flabel returns an encoded form of the target label that can be interpreted by the **_branch** function.

PORTABILITY

_flabel is not portable.

EXAMPLE

```
#include <code.h>
#include <genl370.h>
   /* this code multiplies the integers i and j by   */
   /* repeated addition.  unlike the _blabel example,*/
   /* it behaves correctly when j is less than or    */
   /* equal to 0.  this code has the same effect as  */
   /* the _bfwd example, but uses macros  from       */
   /* genl370.h for readability                      */
_ldregs(R1+R2+R3, i, j, 0);
LTR(2,2);
BC(8, 0, _flabel(2));  /* BZ  LABEL2 */
BC(2, 0, _flabel(1));  /* BP  LABEL1 */
LCR(2,2);
```

```
LCR(1,1);
_label(1);              /* LABEL1 EQU * */
AR(3,1);
BCT(2, 0, _blabel(1)); /* BCT 2,LABEL1 */
_label(2);              /* LABEL2 EQU * */
```

RELATED FUNCTIONS

_bbwd, **_branch**, **_blabel**

_label

Define an Inline Machine Code Branch Target

SYNOPSIS

```
#include <code.h>

void _label(unsigned short n);
```

DESCRIPTION

_label identifies a location in an inline machine code block as a branch target and associates it with an integer **n** between 1 and 65535. **n** must be a compile-time constant. More than one definition using the same integer in a single compilation is permitted. The execution of **_label** has no effect; that is, **_label** defines a location in the object code but does not add any instructions there.

EXAMPLE

See the examples for **_bbwd**, **_bfwd**, **_blabel** and **_flabel**.

Related Functions

_bbwd, **_bfwd**, **_blabel**, **_flabel**.h9e

_ldregs

Load Registers

SYNOPSIS

```
#include <svc.h>

   /* Further arguments are any C expressions - see below. */
void _ldregs(unsigned mask,...);
```

DESCRIPTION

_ldregs is central to the interface between the SAS/C Compiler and inline machine code because it enables you to set up values in machine registers that are used by subsequent machine instructions.

The first argument, **mask**, is a 32-bit mask. This argument must be a compile-time constant because the compiler has to know which registers are needed. General-purpose registers can be specified using the macros R0 through R15, and floating-point registers 0, 2, 4, and 6 can be specified as F0 through F6, respectively. Multiple registers can be specified by adding (or logically ORing) the macros, for example, R0+R1 (or R0|R1). Refer to "Bit Masks for Using Registers" on page 237 for more information on the **mask** argument.

Remaining arguments specify the values to be placed in the registers (low to high) specified by **mask**. Any C expression that has an integer, pointer, or floating-point type may be used. The number of arguments, excluding **mask**, must be equal to the number of one-bits in **mask**; that is, you must supply an expression for each register. The type of each expression must be valid for the register in which the value is to be loaded. For example, the effect of attempting to load a pointer into a floating-point register is unpredictable.

Each call to **_ldregs** starts a new inline machine code sequence. After the call, the contents of any register not explicitly specified in the mask are undefined.

Declarations for **_ldregs** and the associated macros are provided in **<code.h>** as well as **<svc.h>**; either or both can be used.

RETURN VALUE

_ldregs returns no value.

CAUTIONS

You can safely use the following registers:

0 – 3, 14, 15	are general-purpose registers.
0, 2	are floating-point registers.

In addition, you can use floating-point registers 4 and 6 if they are not currently assigned to a register variable.

Do not specify any of the following registers:

4	is used to address constants.
5	is the program base register.
6 – 11	are used for register variables.
12	accesses the C Run-Time Anchor Block (CRAB).
13	addresses the current automatic storage area.

See the register usage warning in "Usage Notes" on page 237.

If you specify incorrect arguments for **_ldregs** (for example, the wrong number of parameters or an invalid register mask), the compiler produces an error message, sets the return code to 8, and generates an EX 0,* instruction for **_ldregs**. The generated instruction causes an execute exception (OS/390 ABEND code 0C3) if it is actually executed.

PORTABILITY

_ldregs is not portable.

IMPLEMENTATION

_ldregs first dumps any values currently in use from the specified registers back to memory. It then clears any information the compiler had about the contents of the

registers. Finally, it loads the registers with the values of the specified C expressions. The order in which the registers are loaded is chosen by the compiler, and it is not necessarily the same as the order of the registers in the mask or the same for every invocation of **_ldregs**.

EXAMPLE

```
#include <regs.h>

   /* Set register 0 to the value 8 and register 14   */
   /* to the address of "field", after first clearing */
   /* any values currently in those registers.        */
long field;

   /* After macro expansion, the final effect of this */
   /* expression is:  _ldregs(0x80020000,8,&field);   */
_ldregs(R0+R14,8,&field);
```

RELATED FUNCTIONS

_cc, **_cms202**, **_code**, **_diag**, **_ossvc**, **_stregs**

_osarmsvc

Generate OS/390 SVC Instruction without Changing Addressing Mode

SYNOPSIS

```
void _osarmsvc(int n);
```

DESCRIPTION

_osarmsvc generates a supervisor call (SVC) instruction. **_osarmsvc** takes one argument (*n*) that specifies the number of the SVC to generate; *n* is an execution-time constant in the range of 0 to 255. No code is generated to change the addressing mode. The SVC will be issued in AR mode if the compiler armode option has been specified, and otherwise will be issued in primary address space mode.

RETURN VALUE

_osarmsvc returns no value. To access values returned in registers, use the **_stregs** function.

CAUTIONS

If you specify incorrect arguments for **_osarmsvc** (such as the wrong number of arguments or an argument that is not a compile-time constant in the range 0 to 255), pass two of the compiler produces an error message, sets the return code to 8, and generates an EX 0,* instruction for **_osarmsvc**. The generated instruction causes an execute exception (OS/390 ABEND code 0C3) if it is executed.

PORTABILITY

_osarmsvc is not portable.

IMPLEMENTATION

_osarmsvc first stores any values currently in use from general-purpose registers 0, 1, 14, and 15 and clears any information the compiler had about the contents of those registers. It assumes that these registers are, or may be, altered by the SVC. **_osarmsvc** then saves the address of the SVC instruction in the C Run-Time Anchor Block (CRAB) as an aid to traceback production in case the SVC causes an abend. Finally, **_osarmsvc** issues the requested SVC instruction.

_OSSVC

Generate OS/390 or CMS SVC Instruction

SYNOPSIS

```
#include <svc.h>

void _ossvc(int n);
```

DESCRIPTION

The **_ossvc** function generates an SVC instruction that is executed in primary address space mode, even if the **armode** compiler option was specified. **_ossvc** takes one argument (**n**) that specifies the number of the SVC to generate; **n** is an execution-time constant in the range of 0 to 255.

RETURN VALUE

_ossvc returns no value. To access values returned in registers, use the **_stregs** function.

CAUTIONS

If you specify incorrect arguments for **_ossvc** (such as the wrong number of arguments or an argument that is not a compile-time constant in the range 0 to 255), pass two of the compiler produces an error message, sets the return code to 8, and generates an EX 0,* instruction for **_ossvc**. The generated instruction causes an execute exception (OS/390 ABEND code 0C3) if it is executed.

If your program executes under CMS, you may want to use the **_cms202** or **_diag** function or both instead of **_ossvc**, depending on the service required.

PORTABILITY

_ossvc is not portable.

IMPLEMENTATION

_ossvc first stores any values currently in use from general-purpose registers 0, 1, 14, and 15 and clears any information the compiler had about the contents of those

registers. It assumes that these registers are, or may be, altered by the SVC. If needed, **_ossvc** then issues a SAC instruction to enter primary address space mode. **_ossvc** then saves the address of the SVC instruction in the C Run-Time Anchor Block (CRAB) as an aid to traceback production in case the SVC causes an ABEND. **_ossvc** then issues the requested SVC instruction. Finally, if the **armode** compiler option is in effect, issue a SAC instruction to restore access register mode.

EXAMPLE

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

   /* Use MVS SVC 11 to obtain the current time as an   */
   /* HHMMSShh GMT value.  Set register 1 to a bit mask */
   /* that requests this form.  HHMMSShh is returned in */
   /* register 0, and a return code is returned in      */
   /* register 15.                                      */
int rc = 0;                   /* return code */
char packed[4] ;       /* area for time in HHMMSShh format */

_ldregs(R1,0x82);             /* Set bit mask in reg 1. */
_ossvc(11);                   /* Issue SVC 11.          */
_stregs(R0+R15,&packed,&rc);  /* returned values        */

if (rc == 0)
printf("GMT is %2x:%02x:%02x:%02x\n",
       packed[0] ,packed[1] ,packed[2] ,packed[3] );

return rc;
```

RELATED FUNCTIONS

_cms202, **_diag**, **_ldregs**, **_stregs**

_stregs

Store Values from Registers

SYNOPSIS

```
#include <svc.h>

   /* Further arguments are expressions */
   /* of pointer type - see below.      */
int _stregs(unsigned mask,...);
```

DESCRIPTION

_stregs enables you to save register values set by an SVC, DIAGNOSE, or machine instruction into memory accessible to the C program. The call to **_stregs** terminates the inline machine code sequence.

The first argument, **mask**, is a 32-bit mask. This argument must be a compile-time constant because the compiler has to know which registers are needed. General-purpose registers can be specified using the macros R0 through R15, and floating-point registers 0, 2, 4, and 6 can be specified as F0 through F6, respectively. Multiple registers can be specified by adding (or logically ORing) the macros, for example, R0+R1 (or R0|R1). Refer to "Bit Masks for Using Registers" on page 237 for more information on the **mask** argument.

Remaining arguments specify where each register is to be stored, in the order (low to high) specified by **mask**. These arguments can be either the address of a variable or a C pointer type. If you specify a pointer, the register is stored in the memory pointed to by the pointer.

If you specify the address of a C variable, the register is stored in that variable. Note that to store a register in a C variable, you must specify the address of the variable. If you specify the variable directly, the variable is treated as a pointer, and the contents of the register are stored in the area addressed by the pointer.

If you specify one more register in the mask than the number of arguments to the function, the value in the highest general-purpose register in the mask is not stored. Instead, it is returned as the value of the **_stregs** function call.

Declarations for **_stregs** and the macros for the register mask can be obtained by including either **<code.h>** or **<svc.h>**.

RETURN VALUE

_stregs returns the value contained in the highest numbered general-purpose register specified in the mask. If no general-purpose register is specified in the mask, the value returned is unpredictable.

To access the condition code set by an SVC, DIAGNOSE, or machine instruction, use the **_cc** function.

CAUTIONS

If you use **_stregs** to obtain the addresses of run-time control blocks (such as the CRAB), code, or data areas, and then modify the contents of the control blocks, code, or data areas, your program is no longer valid C and the results are entirely unpredictable.

Do not use complex expressions, especially ones involving array indexes, as operands of **_stregs** because register shortages can occur. If there is a register shortage, a register whose value was to be stored may be reused. To avoid the problem, assign a complex expression to a pointer variable and specify the variable as the **_stregs** argument. Alternately, you can use the **_code** function to generate ST (store) or STM (store multiple) instructions to store values based on one or more previously loaded addressing registers.

If you specify incorrect arguments for **_stregs** (for example, the wrong number of arguments or an invalid register mask), pass two of the compiler produces an error message, sets the return code to 8, and generates an EX 0,* instruction for **_stregs**. The generated instruction causes an execute exception (OS/390 ABEND code 0C3) if it is actually executed.

PORTABILITY

_stregs is not portable.

IMPLEMENTATION

_stregs stores the values from the specified registers in the places in memory addressed by the expressions. The sequence in which the registers are stored is

determined by the compiler and is not necessarily the same as the order of the registers in the mask. Because the register values are not changed, the compiler retains the information it has about the contents of each register.

EXAMPLE

```
#include <svc.h>

   /* Store the value in register 1 in the C variable reg1, */
   /* store the value in register 15 in the area pointed    */
   /* to by the variable r15 area, and store the value in   */
   /* floating-point register 0 in the area pointed to      */
   /* by the variable dp.                                   */
long reg1;
long  r15area;
double * dp;
_ldregs(0);
   /* After macro expansion, the final effect of this        */
   /* expression is: r15area = _stregs(0x40018000,&reg1,dp); */
r15area = _stregs(R1+R15+F0,&reg1,dp);
```

RELATED FUNCTIONS

_cc, **_cms202**, **_code**, **_diag**, **_ldregs**, **_ossvc**

SVC202, e_SVC202

Generate CMS SVC 202 Instruction with Arguments

SYNOPSIS

```
#include <svc.h>

   /* macro */
int SVC202(r1plist)

   /* macro */
int e_SVC202(r0plist, r1plist)
```

DESCRIPTION

SVC202 and **e_SVC202** (extended SVC 202) generate an SVC 202 instruction. They are intended to be used by programs executing under nonbimodal CMS. **SVC202** takes a pointer to a tokenized PLIST defined in your program. **e_SVC202** takes both a pointer to an untokenized PLIST and a pointer to a tokenized PLIST, both defined in your program.

Note: The **_ossvc** built-in function may be used to generate a SVC 204 in bimodal CMS. △

RETURN VALUE

If no error occurs, these functions return 0. If an error occurs, the value set by the SVC 202 (which is stored in register 15) is returned by the function.

PORTABILITY

SVC202 and **e_SVC202** are not portable.

IMPLEMENTATION

SVC202 and **e_SVC202** are implemented as the following macros:

```
#define SVC202(r1) (_ldregs(R1,r1),_cms202(),_stregs(R15))
#define e_SVC202(r0,r1)
(_ldregs(R0|R1,r0,0x01000000|(unsigned) (r1)),
_cms202(),_stregs(R15))
```

EXAMPLE

Refer to the description of **_cms202** for an alternate version of this example. The VM/XA SP CMS Command Reference contains more details on nucleus extensions.

```
#include <svc.h>
#include <lcstring.h>
#include <stdio.h>

   /* Use the CMS function NUCEXT to determine if the GLOBALV */
   /* command is a nucleus extension.                         */
struct {             /* NUCEXT QUERY parameter list          */
   char cmd[8] ;      /* 'NUCEXT'                             */
   char name[8] ;     /* nucleus extension name               */

   char *scblock;     /* Receives pointer to SCBLOCK.         */
   char *query;       /* 0xffffffff  (identify QUERY)         */
   nucx;
}

int rc;

   /* Copy in the name of the command, padded to           */
   /* eight characters.                                    */
memcpyp(nucx.cmd,"NUCEXT",8,6,' ');

   /* Copy in the name of the nucleus extention            */
   /* to be queried, again padded to eight characters.     */
memcpyp(nucx.name,"GLOBALV",8,7,' ');

nucx.query = (char *) -1;  /* Identify the QUERY function. */

rc=SVC202(&nucx);          /* Issue SVC 202.               */

printf("Globalv %s a nucleus extension\n",
       rc == 0 ? "is" : "is not");
```

RELATED FUNCTIONS

_cms202, **_ossvc**

Macros and Header Files

The inline machine code interface provides several header files to assist in using the built-in functions. Two of these, **<svc.h>** and **<code.h>**, are common to several of the functions. Others are supplied specifically to help simplify issuing machine instructions without having to write your own calls to the **_code** function.

Macros and Header Files for Code Generation

The **_code** header files define macros that can be used to generate specific machine instructions. Each file defines a set of related instructions. For most applications, you need to include only one or two of these files. The header files and a brief description of their contents follow:

<ctl370.h>	problem state program control instructions
<das370.h>	dual address space instructions
<dec370.h>	decimal instructions
<float370.h>	floating-point instructions
<genl370.h>	general-purpose instructions
<io370.h>	370-mode I/O instructions
<ioxa.h>	XA-mode I/O instructions
<lsa370.h>	logical string assist instructions
<str370.h>	string-handling instructions
<supv370.h>	supervisor control instructions
<vec370.h>	vector instructions.

The macros have a format that is similar to assembler language instructions. The following example simply illustrates how to use these macros.

If your program needs to issue the CS instruction, which has two registers and a storage area as operands, use the CS macro. Express the register numbers as integers. Express operands that are storage areas as arithmetic expressions using the macro **b**, which stands for base register. For example, the assembler language instruction

```
CS 14,15,0(1)
```

is written using the CS macro as

```
CS(14,15,0+b(1));
```

The macro expands into the following, which has the same effect as the assembler instruction but is harder to understand:

```
_code(0x00030000, 0xbaef, 0x1000);
```

You should not use register masks in place of register numbers as arguments to these macros. Doing so generates incorrect code or compiler diagnostics, or both.

Note that no macros are provided for instructions that modify registers 4, 5, 12, and 13. Modifying these registers causes subsequent code to fail. The missing instructions are UPT (Update Tree), PC (Program Call), and SIE (Start Interpretive Execution).

The code macros are implemented as calls to other macros, each of which generates a particular instruction format. (For example, the **_RX_** macro generates RX format instructions.) These macros are easy to use and enable you to add special instructions, such as emulation instructions, which may be available at your site.

Details on the macros available in each header file are presented on the following pages. Where header files are too extensive to be shown in their entirety, the beginning and ending instructions are given to provide examples of their format.

<genl370.h>

General-Purpose Instructions Header

This header file defines the general-purpose IBM 370 (and XA) instruction set, except for instructions included in **<str370.h>**, **<dec370.h>**, and **<ctl370.h>** and the UPT instruction, which is omitted because it modifies register 5. This set provides most of the instructions in Chapter 7, "General Instructions," in IBM System/370 Principles of Operation.

In general, you do not need to use the instructions in this set because normal C code can be written to produce identical results with less effort. When you use RX format instructions, note that the index register must be specified as a separate argument from the storage base register. (It should be specified as 0 if no index register is required.)

```
/* ordinary 370 instructions */

#ifndef __Inc_GENL370
#define __Inc_GENL370

#define AR(r1,r2)    _RR_(_R(r1), 0x1a, r1, r2)
#define ALR(r1,r2)   _RR_(_R(r1), 0x1e, r1, r2)
#define NR(r1,r2)    _RR_(_R(r1), 0x14, r1, r2)
#define BALR(r1,r2)  _RR_(_R(r1), 0x05, r1, r2)
#define BASR(r1,r2)  _RR_(_R(r1), 0x0d, r1, r2)
#define BCR(m,r2)    _RR_(0, 0x07, m, r2)
#define BCTR(r1,r2)  _RR_(_R(r1), 0x06, r1, r2)
#define CR(r1,r2)    _RR_(0, 0x19, r1, r2)
#define CLR(r1,r2)   _RR_(0, 0x15, r1, r2)
   .
   .
   .
#define SRL(r1,d)      _RS_(_R(r1), 0x88, r1, 0, d)
#define STCM(r1,m,s)   _RS_(0, 0xbe, r1, m, s)
#define STM(r1,r2,s)   _RS_(0, 0x90, r1, r2, s)
#define CFC(d)         _S_(R1+R2+R3, 0xb21a, d)

   /* UPT not supported due to use of R5
#define UPT()          _E_(R0+R1+R2+R3+R5, 0x0102) */
#define AHI(r1,i2)     _R_(_R(r1),0xa7,r1,0xa,i2)

#endif
```

<str370.h>

String-Handling Instructions Header

This header file defines the instructions CLC, CLCL, CUSE, MVC, MVCIN, MVCL, NC, OC, TR, TRT, and XC.

```
/* 370 string instructions */

#ifndef __Inc_STR370
#define __Inc_STR370

#define CLCL(r1,r2)  _RR_(_RP(r1)+_RP(r2), 0x0f, r1, r2)
#define MVCL(r1,r2)  _RR_(_RP(r1)+_RP(r2), 0x0e, r1, r2)
#define NC(s1,l,s2)    _SS1_(0, 0xd4, l, s1, s2)
#define CLC(s1,l,s2)   _SS1_(0, 0xd5, l, s1, s2)
#define XC(s1,l,s2)    _SS1_(0, 0xd7, l, s1, s2)
#define MVC(s1,l,s2)   _SS1_(0, 0xd2, l, s1, s2)
#define MVCIN(s1,l,s2) _SS1_(0, 0xe8, l, s1, s2)
#define OC(s1,l,s2)    _SS1_(0, 0xd6, l, s1, s2)
#define TR(s1,l,s2)    _SS1_(0, 0xdc, l, s1, s2)
#define TRT(s1,l,s2)   _SS1_(R1 + R2, 0xdd, l, s1, s2)
#define CUSE(r1,r2)  _RRE_((_R(r1)+_R(r2)), 0xb257, r1, r2)

#endif
```

<lsa370.h>

Logical String Assist Instructions

This header file defines the instructions SRST, MVST, and CLST.

```
/* Logical String Assist instructions */
#ifndef __Inc_LSA370
#define __Inc_LSA370

                                  /* SEARCH STRING                 */
#define SRST(r1,r2)  _RRE_((_R(r1)+_R(r2)), 0xb25e, r1, r2)
                                  /* MOVE STRING                   */
#define MVST(r1,r2)  _RRE_((_R(r1)+_R(r2)), 0xb255, r1, r2)
                                  /* COMPARE LOGICAL STRING        */
#define CLST(r1,r2)  _RRE_((_R(r1)+_R(r2)), 0xb25d, r1, r2)
#endif
```

<dec370.h>

Decimal Instructions Header

This header file defines the instructions AP, CP, CVB, CVD, DP, ED, EDMK, MP, MVN, MVO, MVZ, PACK, SP, SRP, UNPK, and ZAP.

```
/* 370 decimal instructions */
#define CVB(r1,x,s)  _RX_(_R(r1), 0x4f, r1, x, s)
#define CVD(r1,x,s)  _RX_(0, 0x4e, r1, x, s)
#define MVN(s1,l,s2)   _SS1_(0, 0xd1, l, s1, s2)
#define MVZ(s1,l,s2)   _SS1_(0, 0xd3, l, s1, s2)
#define ED(s1,l,s2)    _SS1_(0, 0xde, l, s1, s2)
```

```
#define EDMK(s1,l,s2)  _SS1_(R1, 0xdf, l, s1, s2)
#define MVO(s1,l1,s2,l2)  _SS2_(0, 0xf1, l1, l2, s1, s2)
#define PACK(s1,l1,s2,l2) _SS2_(0, 0xf2, l1, l2, s1, s2)
#define UNPK(s1,l1,s2,l2) _SS2_(0, 0xf3, l1, l2, s1, s2)
#define AP(s1,l1,s2,l2)   _SS2_(0, 0xfa, l1, l2, s1, s2)
#define CP(s1,l1,s2,l2)   _SS2_(0, 0xf9, l1, l2, s1, s2)
#define DP(s1,l1,s2,l2)   _SS2_(0, 0xfd, l1, l2, s1, s2)
#define MP(s1,l1,s2,l2)   _SS2_(0, 0xfc, l1, l2, s1, s2)
#define SP(s1,l1,s2,l2)   _SS2_(0, 0xfb, l1, l2, s1, s2)
#define ZAP(s1,l1,s2,l2)  _SS2_(0, 0xf8, l1, l2, s1, s2)
#define SRP(s1,l1,s2,i)   _SS3_(0, 0xf0, l1, i, s1, s2)
```

<float370.h>

Floating-Point Instructions Header

This header file defines the floating-point instruction set, as documented in Chapter 9, "Floating-Point Instructions" in IBM System/370 Principles of Operation.

In general, you do not need to use the instructions in this set because normal C code can be written to produce identical results with less effort. When you use RX format instructions, note that the index register must be specified as a separate argument from the storage base register. It should be specified as 0 if no index register is required.

```
/* This header file defines several esoteric attributes of the         */
/* 370 floating-point implementation.                                  */

#define FLT_RADIX 16         /* hardware float radix                   */
#define FLT_ROUNDS 0         /* float addition does not round          */

#define FLT_MANT_DIG 6       /* hex digits in float mantissa           */
#define DBL_MANT_DIG 14      /* hex digits in double mantissa          */
#define LDBL_MANT_DIG 14     /* hex digits in long double mantissa     */

#define FLT_DIG 6            /* float decimal precision                */
#define DBL_DIG 15           /* double decimal precision               */
#define LDBL_DIG 15          /* long double decimal precision          */

#define FLT_MIN_EXP -64      /* minimum exponent of 16 for float       */
#define DBL_MIN_EXP -64      /* minimum exponent of 16 for double      */
#define LDBL_MIN_EXP -64     /* minimum exponent of 16 for long double */

#define FLT_MIN_10_EXP -78   /* minimum float power of 10              */
#define DBL_MIN_10_EXP -78   /* minimum double power of 10             */
#define LDBL_MIN_10_EXP -78  /* minimum long double power of 10        */

#define FLT_MAX_EXP 63       /* maximum exponent of 16 for float       */
#define DBL_MAX_EXP 63       /* maximum exponent of 16 for double      */
#define LDBL_MAX_EXP 63      /* maximum exponent of 16 for long double */

#define FLT_MAX_10_EXP 75    /* maximum float power of 10              */
#define DBL_MAX_10_EXP 75    /* maximum double power of 10             */
#define LDBL_MAX_10_EXP 75   /* maximum long double power of 10        */

#define FLT_MAX .7237005e76F             /* maximum float              */
#define DBL_MAX .72370055773322621e76    /* maximum double             */
```

```
#define LDBL_MAX .72370055773322621e76L  /* maximum long double        */

#define FLT_EPSILON .9536743e-6F             /* smallest float x such
                                                that 1.0 + x != 1.0    */
#define DBL_EPSILON .2220446049250313le-15   /* smallest double x such
                                                that 1.0 + x != 1.0    */
#define LDBL_EPSILON .2220446049250313le-15L /* smallest long double x such
                                                that 1.0 + x != 1.0    */

#define FLT_MIN .5397606e-78F            /* minimum float              */
#define DBL_MIN .53976053469340279e-78   /* minimum double             */
#define LDBL_MIN .53976053469340279e-78L /* minimum long double        */
```

<ctl370.h>

Problem Program Control Instructions

This header file defines the instructions BASSM, BSM, CDS, CS, DIAG, EX, IPM, MC, SPM, STCK, SVC, and TS. Because the DIAG, EX, MC, and SVC instructions may have varying effects depending on their operands and environment, these macros have a register mask as an additional final operand. For example, **`MC(540,0x80,R1+R15)`** generates the same code as the assembler instruction MC 540,X'80' and informs the compiler that the contents of registers 1 and 15 may be changed. **<ctl370.h>** also includes an **`EX_SS`** macro, to simplify using the EX instruction to execute an SS-format instruction. For example, **`EX_SS(14,TR(0+b(15),0,0+b(2)))`** generates code equivalent to the assembler sequence, as shown here:

```
BALR  1,0
        B     EXINSTR
TARGET  TR    0(0,15),0(2)
EXINSTR EX    14,TARGET
```

Note that this macro uses register 1 as a work register. To use **`EX_SS`**, you must include **<genl370.h>** in addition to **<ctl370.h>** and the header file containing the target of the EX instruction.

```
/* program control instructions */

#ifndef __Inc_CTL370
#define __Inc_CTL370

#define BASSM(r1,r2) _RR_(_R(r1), 0x0c, r1, r2)
#define BSM(r1,r2)   _RR_(_R(r1) & ~R0, 0x0b, r1, r2)
#define SPM(r1)      _RR_(_R(r1), 0x04, r1, 0)
#define EX(r1,x,s,m) _RX_(m, 0x44, r1, x, s)
#define EX_SS(r1,instr) (BAL(1,0,_flabel(0xeeee)),
                         instr,
                         _label(0xeeee),
                         EX(r1,0,0+b(1),0))
#define MC(s,i,m)    _SI_(m, 0xaf, i, s)
#define SVC(i,m)   _I_(m, 0x0a, i)
#define DIAG(r1,r2,i,m)  _RRI_(m, 0x83, r1, r2, i)
#define CS(r1,r2,s)    _RS_(_R(r1)+_R(r2), 0xba, r1, r2, s)
#define CDS(r1,r2,s)   _RS_((_RP(r1))+_RP(r2),0xbb,r1,r2,s)
#define IPM(r1)      _RRE_(_R(r1), 0xb222, r1, 0)
#define STCK(s)  _S_(0, 0xb205, s)
```

```
#define TS(s)    _S_(0, 0x9300, s)
#define TRAP2() _E_(0,0x01ff)
#define TRAP4() _S_(0, 0xb2ff, s)

#endif
```

<supv370.h>

Supervisor Control Instructions Header

This header file defines all the instructions defined in Chapter 10, "Control Instructions" in IBM System/370 Principles of Operation, except those defined in **<ctl370.h>** and **<das370.h>**. Both IBM 370 and XA specific instructions are included.

```
/* 370 supervisor control instructions */

#ifndef __Inc_SUPV370
#define __Inc_SUPV370

#define ISK(r1,r2)   _RR_(_R(r1), 0x09, r1, r2)
#define SSK(r1,r2)   _RR_(0, 0x08, r1, r2)
#define LRA(r1,x,s)  _RX_(_R(r1), 0xb1, r1, x, s)
#define RDD(s,i)     _SI_(0, 0x85, i, s)
#define STNSM(s,i)   _SI_(0, 0xac, i, s)
#define STOSM(s,i)   _SI_(0, 0xad, i, s)
#define WRD(s,i)     _SI_(0, 0x84, i, s)
#define MVCK(s1,r1,s2,r3) _SK_(0, 0xd9, r1, r3, s1, s2)
#define LCTL(r1,r2,s)  _RS_(0, 0xb7, r1, r2, s)
#define SIGP(r1,r2,d)  _RS_(_R(r1), 0xae, r1, r2, d)
#define STCTL(r1,r2,s) _RS_(0, 0xb6, r1, r2, s)
#define TRACE(r1,r2,s) _RS_(0, 0x99, r1, r2, s)
#define RRBE(r2)     _RRE_(0, 0xb22a, 0, r2)
#define ISKE(r1,r2)  _RRE_(_R(r1), 0xb229, r1, r2)
#define IVSK(r1,r2)  _RRE_(_R(r1), 0xb223, r1, r2)
#define IPTE(r1,r2)  _RRE_(0, 0xb221, r1, r2)
#define SSKE(r1,r2)  _RRE_(0, 0xb22b, r1, r2)
#define TB(r2)       _RRE_(R0, 0xb22c, 0, r2)
#define TPROT(s,d) _SSE_(0, 0xe501, s, d)
#define IPK()    _S_(R2, 0xb20b, 0)
#define LPSW(s)  _S_(0, 0x8200, s)
#define PTLB()   _S_(0, 0xb20d, 0)
#define RRB(s)   _S_(0, 0xb213, s)
#define SCK(s)   _S_(0, 0xb204, s)
#define SCKC(s)  _S_(0, 0xb206, s)
#define SPT(s)   _S_(0, 0xb208, s)
#define SPX(s)   _S_(0, 0xb210, s)
#define SPKA(d)  _S_(0, 0xb20a, d)
#define SSM(s)   _S_(0, 0x8000, s)
#define STCKC(s) _S_(0, 0xb207, s)
#define STAP(s)  _S_(0, 0xb212, s)
#define STIDP(s) _S_(0, 0xb202, s)
#define STPT(s)  _S_(0, 0xb209, s)
#define STPX(s)  _S_(0, 0xb211, s)
/* SIE not supported due to use of R4/R5/R12/R13
#define SIE(s)   _S_(R0+R1+R2+R3+R4+R5+R6+R7+R8+R9+R10+R11+R12+R13,
```

```
                    0xb214, s) */
#define MVPG(r1,r2)  _RRE_(0, 0xb254, r1, r2)
#endif
```

<das370.h>

Dual Address Space Instruction Header

This header file defines the instructions EPAR, ESAR, IAC, LASP, MVCP, MVCS, PT, SAC, SSAR and SACF. (PC is omitted because it modifies register 4, which is not permitted by the compiler.)

```
/* 370 dual address space instructions */

#ifndef __Inc_DAS370
#define __Inc_DAS370

#define MVCP(s1,r1,s2,r3) _SK_(0, 0xda, r1, r3, s1, s2)
#define MVCS(s1,r1,s2,r3) _SK_(0, 0xdb, r1, r3, s1, s2)
#define EPAR(r1)     _RRE_(0, 0xb226, r1, 0)
#define ESAR(r1)     _RRE_(0, 0xb227, r1, 0)
#define IAC(r1)      _RRE_(0, 0xb224, r1, 0)
#define PT(r1,r2)    _RRE_(0, 0xb228, r1, r2)
#define SSAR(r1)     _RRE_(0, 0xb225, r1, 0)
#define LASP(s,d)  _SSE_(0, 0xe500, s, d)
/* PC not supported due to use of R4
#define PC(d)    _S_(R3+R4+R14, 0xb218, d) */
#define SAC(d)   _S_(0, 0xb219, d)
#define SACF(d)  _S_(0, 0xb2719, d)

#endif
```

<io370.h>

370-Mode I/O Instructions Header

This header file defines all the I/O instructions listed in the "Input/Output Operations" chapter of IBM System/370 Principles of Operation.

```
/* 370 I/O instructions */

#define CONCS(d) _S_(0, 0xb200, d)
#define DISCS(d) _S_(0, 0xb201, d)
#define CLRCH(d) _S_(0, 0x9f01, d)
#define CLRIO(d) _S_(0, 0x9d01, d)
#define HDV(d)   _S_(0, 0x9e01, d)
#define HIO(d)   _S_(0, 0x9e00, d)
#define SIO(d)   _S_(0, 0x9c00, d)
#define SIOF(d)  _S_(0, 0x9c01, d)
#define RIO(d)   _S_(0, 0x9c02, d)
#define STIDC(d) _S_(0, 0xb203, d)
#define TCH(d)   _S_(0, 0x9f00, d)
#define TIO(d)   _S_(0, 0x9d00, d)
```

<ioxa.h>

XA-Mode I/O Instructions Header

This header file defines all the I/O instructions listed in Chapter 13, "I/O Instructions" in IBM SYSTEM/370 XA Principles of Operation.

```
/* XA I/O instructions */

#define CSCH()   _S_(0, 0xb230, 0)
#define HSCH()   _S_(0, 0xb231, 0)
#define MSCH(s)  _S_(0, 0xb232, s)
#define RCHP()   _S_(0, 0xb23b, 0)
#define RSCH()   _S_(0, 0xb238, 0)
#define SAL()    _S_(0, 0xb237, 0)
#define SCHM()   _S_(0, 0xb23c, 0)
#define SSCH(s)  _S_(0, 0xb233, s)
#define STCPS(s) _S_(0, 0xb23a, s)
#define STCRW(s) _S_(0, 0xb239, s)
#define STSCH(s) _S_(0, 0xb234, s)
#define TPI(s)   _S_(0, 0xb236, s)
#define TSCH(s)  _S_(0, 0xb235, s)
```

<vec370.h>

Vector Instructions Header

This header file defines all the vector instructions defined in IBM System/370 Vector Operations. Note that the compiler never accesses or modifies vector registers, so any use of these registers must be done using assembler language subroutines, these macros, or the **_code** function.

```
/* 370 vector instructions */

#define VCVM()       _RRE_(0, 0xa641, 0, 0)
#define VCZVM(r1)    _RRE_(_R(r1), 0xa642, r1, 0)
#define VCOVM(r1)    _RRE_(_R(r1), 0xa643, r1, 0)
#define VXVC(r1)     _RRE_(_R(r1), 0xa644, r1, 0)
#define VXVMM(r1)    _RRE_(_R(r1), 0xa646, r1, 0)
#define VLVCU(r1)    _RRE_(_R(r1), 0xa645, r1, 0)
#define VRRS(r1)     _RRE_(_RP(r1), 0xa648, r1, 0)
#define VRSVC(r1)    _RRE_(_RP(r1), 0xa649, r1, 0)
#define VRSV(r1)     _RRE_(_RP(r1), 0xa64a, r1, 0)
#define VTVM()       _RRE_(0, 0xa640, 0, 0)
#define VRCL(d)  _S_(0, 0xa6c5, d)
   .
   .
   .
#define VMNSE(r1,r3,r2) _VR_(_FR(r3)|(_RP(r2) & ~(R0+R1)),
                            0xa601, r3, r1, r2)
#define VSPSD(r1,r2)    _VR_(_FR(r2), 0xa61a, 0, r1, r2)
#define VZPSD(r1)       _VR_(0, 0xa61b, 0, r1, 0)
#define VLBIX(r1,r3,s) _RSE_(_RP(r3), 0xe428, r3, r1, s)
#define VLI(r1,r3,s)   _RSE_(0, 0xe400, r3, r1, s)
#define VLID(r1,r3,s)  _RSE_(0, 0xe410, r3, r1, s)
#define VLIE(r1,r3,s)  _RSE_(0, 0xe400, r3, r1, s)
```

```
#define VSLL(r1,r3,d)  _RSE_(0, 0xe425, r3, r1, d)
#define VSRL(r1,r3,d)  _RSE_(0, 0xe424, r3, r1, d)
#define VSTI(r1,r3,s)  _RSE_(0, 0xe401, r3, r1, s)
#define VSTID(r1,r3,s) _RSE_(0, 0xe411, r3, r1, s)
#define VSTIE(r1,r3,s) _RSE_(0, 0xe401, r3, r1, s)
```

General Header Files

The header files **<code.h>**, **<regs.h>**, and **<svc.h>** provide symbolic definitions and condition code values used by the built-in functions. The contents of these files are as shown on the following pages.

<code.h>

Code Header File The **<code.h>** header file contains declarations needed by the macros and the **_code** function.

```
#ifndef __IncCode
#define __IncCode

#ifndef __IncRegs
#define __IncRegs
#include <sys370/regs370.h>
#endif


#ifdef __cplusplus
extern "C" {
void __builtin _code(unsigned,...);
void __builtin _label(unsigned);
void __builtin _branch(unsigned,unsigned,unsigned);
void __builtin _bbwd(unsigned,unsigned);
void __builtin _bfwd(unsigned,unsigned);
}
#else
#define _code __builtin_code
#define _label __builtin_label
#define _branch __builtin_branch
#define _bbwd __builtin_bbwd
#define _bfwd __builtin_bfwd
#ifndef _NOLIBCK
void _code(unsigned,...);
void _label(unsigned);
void _branch(unsigned,unsigned,unsigned);
void _bbwd(unsigned,unsigned);
void _bfwd(unsigned,unsigned);
#else
void _code();
void _label();
void _branch();
void _bbwd();
void _bfwd();
#endif
#endif
```

```
#define b(n) ((n) << 12)
#define _R(n) (R0 >> (n))
#define _RP(n) ((R0+R1) >> (n))
#define _FR(n) (F0 >> (n))
#define _FRP(n) ((F0+F2) >> (n))
#define _seg(x) ((int) R0 >> (x))
#define _blabel(n) ((unsigned short)(n) + 0x00bb0000)
#define _flabel(n) ((unsigned short)(n) + 0x00bf0000)

#define _RR_(m,o,r1,r2) _code(m, (o)<<8|(r1)<<4|(r2))
#define _RX_(m,o,r1,r2,bd) _code(m, (o)<<8|(r1)<<4|(r2), bd)
#define _BX_(m,o,r1,r2,bd) _branch(m, (o)<<8|(r1)<<4|(r2), bd)
#define _SI_(m,o,i,bd) _code(m, (o)<<8|(i),bd)
#define _SS1_(m,o,l,bd1,bd2) _code(m, (o)<<8|(l?(l)-1:l), bd1, bd2)
#define _SS2_(m,o,l1,l2,bd1,bd2) _code(m, (o)<<8|(l1?(l1)-1:l1)<<4|
                                    (l2?l2-1:l2), bd1, bd2)
#define _SS3_(m,o,l1,i,bd1,bd2) _code(m, (o)<<8|(l1?(l1)-1:l1)<<4|(i),
                                   bd1, bd2)
#define _SK_(m,o,r1,r3,bd1,bd2) _code(m, (o)<<8|(r1)<<4|(r3), bd1, bd2)
#define _I_(m,o,i) _code(m, (o)<<8|(i))
#define _RRI_(m,o,r1,r2,i) _code(m, (o)<<8|(r1)<<4|(r2), i)
#define _RS_(m,o,r1,r2,bd) _code(m, (o)<<8|(r1)<<4|(r2), bd)
#define _BS_(m,o,r1,r2,bd) _branch(m, (o)<<8|(r1)<<4|(r2), bd)
#define _RRE_(m,o,r1,r2) _code(m, o, (r1)<<4|(r2))
#define _S_(m,o,bd) _code(m, o, bd)
#define _E_(m,o) _code(m, o)
#define _VST_(m,o,r3,t2,r1,s2) _code(m, o, (r3)<<12|(t2)<<8|
                                  (r1)<<4|(s2))
#define _VV_(m,o,r3,r1,r2) _code(m, o, (r3)<<12|(r1)<<4|(r2))
#define _QST_(m,o,r3,t2,r1,s2) _code(m, o, (r3)<<12|(t2)<<8|
                                  (r1)<<4|(s2))
#define _QV_(m,o,r3,r1,r2) _code(m, o, (r3)<<12|(r1)<<4|(r2))
#define _VS_(m,o,r2) _code(m, o, r2)
#define _VR_(m,o,r3,r1,r2) _code(m, o, (r3)<<12|(r1)<<4|(r2))
#define _RSE_(m,o,r3,r1,bd) _code(m, o, (r3)<<12|(r1)<<4, bd)

#endif
```

<regs.h>

Register Values Header File

The **<regs.h>** header file contains the symbolic definitions and condition codes used by the **_cc**, **_ldregs**, and **_stregs** functions.

```
#define R0 0x80000000
#define R1 0x40000000
#define R2 0x20000000
#define R3 0x10000000
#define R6 0x02000000
#define R7 0x01000000
#define R8 0x00800000
#define R9 0x00400000
#define R10 0x00200000
```

```
#define R11 0x00100000
#define R12 0x00080000
#define R13 0x00040000
#define R14 0x00020000
#define R15 0x00010000
#define F0 0x00008000
#define F2 0x00002000
#define F4 0x00000800
#define F6 0x00000200

#define CC0 0
#define CC1 (-1)
#define CC2 2
#define CC3 3
#define CCZ 0
#define CCE 0
#define CCL (-1)
#define CCM (-1)
#define CCH 2
#define CCP 2
#define CCO 3

#ifdef __cplusplus
extern "C" {
extern void __builtin _ldregs(unsigned,...);
extern int __builtin _stregs(unsigned,...);
extern int __builtin _cc(void);
}
#else
#define _ldregs __builtin_ldregs
#define _stregs __builtin_stregs
#define _cc     __builtin_cc

#ifndef _NOLIBCK
extern void _ldregs(unsigned,...);
extern int _stregs(unsigned,...);
extern int _cc(void);
#else
extern void _ldregs();
extern int _stregs();
extern int _cc();
#endif
#endif
```

<svc.h>

Supervisor Control Header File

The **<svc.h>** header file contains the symbolic definitions and condition codes used by the **_cms202**, **_diag**, and **_ossvc** functions.

```
#ifndef __IncSvc
#define __IncSvc

#ifndef __IncRegs
```

```
#define __IncRegs
#include <regs.h>
#endif

#define _ossvc  __builtin_ossvc
#define _cms202 __builtin_cms202
#define _diag   __builtin_diag

#define SVC202(r1) (_ldregs(R1,r1),_cms202(),_stregs(R15))
#define e_SVC202(r0,r1)
(_ldregs(R0|R1,r0,0x01000000|(unsigned) (r1)),_cms202(),_stregs(R15))

#ifndef _NOLIBCK
extern void _ossvc(int);
extern void _cms202(void);
extern void _diag(int);
#else
extern void _ossvc();
extern void _cms202();
extern void _diag();
#endif
```

.h9e

Example of the Inline Machine Code Interface

This example is a C implementation of the compare and swap example in IBM System/370 Principles of Operation. Register conventions have been changed to conform to C usage. (The original example used registers 6, 7, and 8, which may be allocated to **`register`** variables in C.)

```
#include <code.h>
#include <genl370.h>
#include <ctl370.h>

int word;                    /* the flag word */
char flag = 0x80;            /* the bit to be turned on */

/* put word in R0, flag in R14 and        */
/* &word in R1.  word is loaded only      */
/* once to avoid inconsistent results     */
/* if it is updated by another processor. */
retry:
   _ldregs(R0+R1+R14, word, &word, flag << 24);

   LR(15,0);                    /* copy word to R15 */
   OR(15,14);                   /* turn on the flag */
   CS(0,15,0+b(1));             /* try to update the word*/

   if (_cc() != 0)              /* try again if swap failed */
      goto retry;
```

This version of the example performs an unnecessary reload of registers in the unlikely event that CS returns a non-zero condition code. The unnecessary reload

cannot be corrected by attaching the **retry** label to the LR instruction because inserting the label between the call to **_ldregs** and the LR instruction may cause one of the registers loaded by **_ldregs** to be updated. However, the following code sequence does bypass the problem for a slight performance improvement in the event that CS returns a value other than 0.

```
#include <code.h>
   #include <genl370.h>
   #include <ctl370.h>

   int word;                     /* the flag word */
   char flag = 0x80;             /* the bit to be turned on */

      /* Put word in R0, flag in R14 and        */
      /* &word in R1.  word is loaded only      */
      /* once to avoid inconsistent results     */
      /* if it is updated by another processor. */
   _ldregs(R0+R1+R14, word, &word, flag << 24);

   BALR(2,0);                    /* set up a base register      */
   LR(15,0);                     /* copy word to R15            */
   OR(15,14);                    /* turn on the flag            */
   CS(0,15,0+b(1));              /* try to update the word      */
   BCR(7,2);                     /* try again if swap failed    */
```

CHAPTER

14

Systems Programming with the SAS/C Compiler

Introduction

This chapter discusses the C Systems Programming Environment (SPE). SPE is an implementation of the C execution framework that is designed to be used for IBM 370 operating systems programming.

Intended Audience

This chapter is written for systems programmers experienced with the IBM 370 OS/390 operating system, the CMS component of VM, or the CICS teleprocessing monitor. No attempt is made to explain terms and concepts that are generally used in systems programming. Also, this section assumes that you are familiar with assembler language programming, programming using the C language in general, and the SAS/C implementation of the C language, specifically.

Related Documentation

Refer to the following chapters in this book for more information:

- Chapter 3, “Code Generation Conventions,” on page 45 for information about how the compiler creates object code.
- Chapter 6, “Compiler Options,” on page 101 for information about the **`indep`** compiler option.
- Chapter 11, “Communication with Assembler Programs,” on page 209 for general information about C assembler language interfaces.
- Chapter 13, “Inline Machine Code Interface,” on page 231 contains information about the built-in functions used in the SPE library macros.
- Appendix 1, “The DSECT2C Utility,” on page 343 describes DSECT2C, a program that converts assembler language DSECTs to C structure definitions.

The SAS/C Library Reference, Volume 2 also contains the following related information:

- Chapter 2, "CMS Low-Level I/O Functions"
- Chapter 3, "OS/390 Low-Level I/O Functions"
- Chapter 4, "OS/390 Multitasking and Other Low-Level System Interfaces"
- Chapter 19, "Introduction to POSIX"

Also, refer to the appropriate IBM documentation for your operating system.

Source Code Files

The programmer can modify the SPE library. Therefore, all of the elements that interact with the operating system are delivered in both source code and object code

format. Also, sample programs are included as part of SPE. These samples are intended merely to illustrate SPE programming techniques but can in some cases be useful in particular applications.

The names of the samples and the source and object code files begin with the characters L$U. The source code can be found in the SASC.SOURCE data set (under OS/390) or the LSU MACLIB (under CMS). The object code is in SASC.SPEOBJ (under OS/390) or LC370SPE TXTLIB (under CMS). The object code for CICS is in SASC.CICS.SPEOBJ (under OS/390) or LC370SPC TXTLIB (under CMS). Ask your SAS Software Representative for C compiler products for more information about these files.

Assembler language macros

The assembler language macros used in SPE source code files can be found in the SASC.MACLIBA data set (under OS/390) or in LCUSER MACLIB (under CMS). The ABORT, GETSTG, FREESTG, LOADM, and UNLOADM macros accept a SYS= parameter to indicate the target operating system. If SYS=CMS is specified, the macro expands to use a standard CMS interface (such as DMSFREE). If SYS=BI is specified, the macro expands to use a CMS XA interface (such as CMSSTOR). If you specify CICS, the macro expands to use a CICS interface, such as EXEC CICS GETMAIN. Any other parameter causes the macro to use an OS/390 interface (such as GETMAIN). The default value depends on the operating system under which the program is assembled, either OS/390 or CMS. The LOADM and UNLOADM macros are not supported for CICS. Instead, code the corresponding EXEC CICS LOAD and RELEASE commands; the SPE **`loadm`** and **`unloadm`** functions are fully supported for CICS.

Code generation for XA and 370 Mode CMS

Some SPE C header and source files need to generate different code in XA and 370 mode CMS. These files test whether the symbol BIMODAL is defined in order to decide what code to generate.

An Overview of SPE

The C Language and Systems Programming

The C language was originally designed for general purpose programming under UNIX operating systems. Although C was used for writing the operating system itself, the language design allowed C language programs to be relatively portable across widely disparate operating systems and machine architectures. The portability of C has recently been enhanced by the efforts of the ANSI X3J11 Committee to standardize the C language, including the run-time library.

In the systems programming environment, however, portability is not an important issue. Systems programs cannot restrict themselves to a portable view of the system. An implementation of C for this environment must provide access to system data and services without regard for other environments. It must be efficient. Above all, it must be adaptable to the requirements of the operating system.

SPE is designed to enable the C language to be used as a systems programming language in the IBM 370 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.

SPE is designed to enable C programs to interface with the operating system in the same way an assembler language program does. For example, storage can be allocated

with a DMSFREE or GETMAIN macro, each of which expands to the appropriate SVC instruction. SPE programs can access parameter lists like assembler language programs do. Under CMS, the SPE equivalent of the **main** function gets pointers to the tokenized and untokenized PLISTs as arguments, instead of using a portable format such as **argv** and **argc**.

Adapting SPE

All of the SPE routines that interface with the operating system are provided in source code. Each routine provides basic functionality in a generic manner. However, given the wide variety of unique SPE applications, no single interface can address all the conflicting needs efficiently. Therefore, these routines can be (and are expected to be) adapted as necessary for the needs of the environment. For example, if the implementation of the stack in L$UPROL is not practical in a particular environment, then it can be redesigned for that environment.

"The SPE Framework: Creating and Terminating" on page 280 and "SPE Internals" on page 289 contain detailed information on how each of the major SPE routines operates and what is expected of them. Each discussion is oriented toward the role the routine plays in supporting the SPE execution framework and how this framework interacts with compiler-generated code.

The Run-Time Library

The SPE run-time library is designed to support systems programs in the C language. The library contains three classes of functions. In the first class are those functions that interface directly to the operating system. For example, instead of an **fopen** function, SPE provides a function to open an OS/390 data set using BSAM and a function to open a CMS file with FSOPEN. In the second class are those functions that are present in the full C library but do not interface to the operating system, such as math and string functions. In the third class are those functions, such as **malloc**, that are present in the full C library but have operating system dependencies. The SPE library contains SPE versions of these functions that differ from the full library functions.

The SPE library also supports many POSIX functions, such as **open**, **chdir**, and **fork**. Most of the POSIX functions are implemented within the OS/390 control program, and support for these functions is therefore no different from the SPE support for BSAM and GETMAIN. POSIX functions that are not supported directly by UNIX System Services (USS) OS/390, such as **fdopen** and **tzset**, are in general not supported with SPE.

"The SPE Library" on page 300 contains descriptions of all the functions that are designed for use with SPE. Each description also explains how the function is implemented. When the function interfaces with the operating system, the description gives the name of the associated source code file. In many cases, common adaptations are mentioned, along with suggestions as to how they can be implemented.

One of the most frequent applications in systems programming is interrupt handling. "Interrupt Handling in SPE" on page 295 describes the **bldexit** function, which provides an interface between an operating system interrupt exit and a C program. **bldexit** is a prototypical SPE function in that it forsakes portability for adaptability and efficiency.

SPE and the Debugger

The SPE framework does not include support for the debugger.

The SPE Framework: Creating and Terminating

When a C program is invoked, the C execution framework is created by a start-up routine that is executed before the **main** function is called. When **main** returns or when the **exit** function is called, the framework is destroyed. The full C framework provides the support environment needed by the full C library, including I/O, command-line parsing, signal handling, stack and heap storage management, dynamic loading, and many other services. In ANSI terminology, this is called a hosted environment.

In SPE, the framework is created by a start-up routine that can be modified to allow a framework to be created in any number of ways. The framework itself can be tailored to support whatever services are needed by the program. In ANSI terms, this is called a freestanding environment.

In general, there are three methods that you can use to create an SPE framework:

- Use one of the standard OS/390, CMS, or CICS start-up routines. These start-up routines are intended to be simple and generic. Their design is heavily biased toward traditional assembler language concepts. The framework is destroyed when this routine returns or when **exit** is called.

  These start-up routines are most appropriate for straightforward applications consisting of a main program and subroutines. The framework can be modified to be as simple or as complex as necessary.
- Create an INDEP framework. With this type of framework, the framework is created when the program is first entered but remains in existence even after control is transferred back to the caller (usually the operating system) by returning from the initial function. The framework can be destroyed by calling **exit** or a special routine called L$UEXIT. See "The L$UEXIT Routine" on page 282 for more information.

  This method is most appropriate for applications structured as a package of subroutines without a main routine. The first subroutine called creates the framework, which is thereafter shared between all subroutines. Each function compiled using the **indep** option can be separately invoked and can return control independently to the operating system (or whatever program called it) without destroying the C framework.
- Write your own start-up routine. These start-up routines are tailored to a specific application or set of applications. Each start-up routine can call a standard subroutine to create and destroy the C framework. This subroutine can, in turn, be tailored as necessary to create a framework with the required support.

  This method is most appropriate for applications with unusual linkage requirements, such as having arguments in unusual registers or not having a valid save area pointer in register 13.

Of course, all three methods can be combined. The needs of the application may be such that the standard start-up routines can be slightly modified to provide for them. You may want to write your own start-up routine that creates an INDEP framework. SPE is designed to be flexible.

No matter which method is chosen, most of the details of creating and destroying frameworks are handled by a routine called L$UMAIN. In this section, L$UMAIN is described first because it is used in all of the methods. After this discussion, each of the three methods of creating an SPE framework is discussed individually.

The L$UMAIN Routine

The compiler generates code that depends on the following conventions. L$UMAIN ensures that these conventions are met when it calls the initial functions.

R1	addresses the function's parameter list.
R12	addresses the CRAB.
R13	addresses a DSA.
R14	contains the function's return address.
R15	contains the function's entry point.
CRABPRV	addresses the pseudoregister vector.

When L$UMAIN is entered, certain values must be in the general registers, as follows:

R1	points to a word in which the address of the CRAB will be stored. If R1 is 0, then the CRAB address will not be stored.
R13	points to a save area in which the registers have already been saved using a **`STM 14,12,12(13)`** instruction. The R15 slot of this save area (offset X'10') must address the initial function to be executed under the C framework. The R1 slot (offset X'18') must contain the value that is to be placed in R1 when the initial function is called. This value will be used as the address of the initial function's parameter list. The initial function can be either a C function or an assembler function that uses the CENTRY and CEXIT macros.
R14	contains the return address.
R15	contains the address of L$UMAIN.

L$UMAIN performs the following steps:

1. allocates storage for the CRAB and initializes it.
2. saves the R13 value on entry in the CRABPENV field of the CRAB.
3. allocates storage for the pseudoregister vector.
4. calls the L$CPRSU routine to initialize the pseudoregister vector with initial values for **`extern`** and **`static`** variables.
5. allocates the C stack. The stack size can be specified as the initial value of the **`extern`** variable **`_stack`**. (Note that the PRV is initialized before the stack is created, so the initial value of **`_stack`** will be available.)
6. points R13 to the first save area on the stack and chains it to the caller's save area. L$UMAIN also saves this address in the CRABMDSA field in the CRAB.
7. calls the function whose address was in the R15 slot of the save area on entry.

Note that you can modify L$UMAIN to add, modify, or remove steps as required by the application. Creation of a CRAB and a stack is always required.

L$UMAIN calls the initial function using a BALR 5,15 instruction. This places the return address for L$UMAIN in R5. The return address for L$UMAIN's caller (the start-up routine) remains in R14. This enables the first C function to return directly to the start-up routine.

The start-up routine then has the following options:

Option 1	Branch to the address in R5, thereby returning to L$UMAIN. This path enables L$UMAIN to terminate the C framework and return to the start-up routine's caller.
Option 2	Load R13 from the chain field of L$UMAIN's save area, restore registers, and return to the address in R14. This path bypasses C framework termination. The save area for L$UMAIN remains allocated, and the C framework continues to exist. You can destroy the framework later by calling **`exit`** or L$UEXIT.

The L$UEXIT Routine

Termination of the C framework occurs when control returns to L$UMAIN. In most frameworks, control returns to L$UMAIN when the initial function returns or when the **exit** function is called.

If a program is entered via an INDEP function, L$UEXIT can be called as a function to terminate the framework. L$UEXIT forces a return to L$UMAIN by loading R13 from the CRABMDSA field (thus addressing L$UMAIN's save area), reloading L$UMAIN's registers from this save area, and branching to the address in R5.

L$UMAIN performs the following steps to terminate the C framework:

1. calls any defined **atexit** cleanup routines.
2. loads R13 from the CRABPENV field. Normally, this addresses the save area belonging to the caller of the start-up routine.
3. frees any memory allocated for library control blocks, including heap and stack storage, the pseudoregister vector, and the CRAB.
4. restores registers and returns. Note that control returns to the caller of the start-up routine and not to the start-up routine itself.

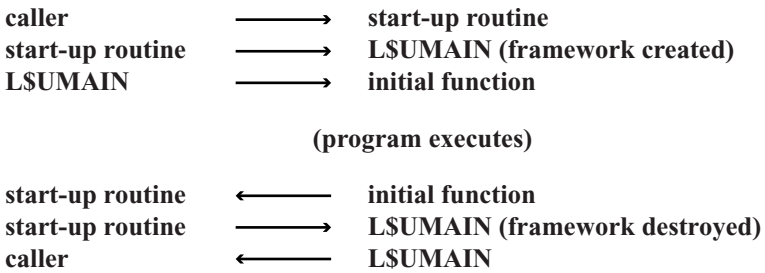
L$UEXIT can be called from a non-C routine. The **exit** function calls L$UEXIT, but the **exit** function is designed to be called only from a C function. Note that L$UEXIT can be used only in an INDEP framework.

The Standard Start-up Routines

The standard start-up routines are provided as generic examples. For programs running under OS/390, the standard start-up routine is named L$UOSEP. For CMS programs, the standard start-up routine is named L$UCMSE. For USS, the standard start-up routine is L$UOEEP. The standard start-up routine for CICS is named L$UCICE. These routines are written to provide a minimal environment with linkage and parameters that are appropriate for the operating system.

The general flow of control of these routines is shown in Figure 14.1 on page 282. The start-up routines create the framework by calling L$UMAIN and destroy it by allowing control to return to L$UMAIN after the initial function returns.

Figure 14.1 Standard Start-up Flow of Control



Do the following to use the standard SPE start-up routine:

- Include the appropriate header file:
  - **<osmain.h>** for OS/390
  - **<oemain.h>** for USS OS/390

- **<cmsmain.h>** for CMS
- **<cicsmain.h>** for CICS.

These header files define symbols that force the linkage editor to include the correct start-up routine (L$UOSEP, L$UOEEP, L$UCMSE, or L$UCICE) and select the correct entry point. The entry point for OS/390 is #OSEP; for USS OS/390, #OEEP; for CMS, #CMSEP; for CICS, #CICSEP.

- Define an initial function named **oemain**, **osmain**, **cmsmain**, or **cicsmn**. This function is equivalent to the **main** function in the full C framework. An individual description for each system's function is described in the next sections.

The standard OS/390 start-up routine

The standard OS/390 start-up routine L$UOSEP expects to be entered with R1 addressing a standard OS VL-format parameter list containing the addresses of one or more parameters, with the last address indicated by the presence of the VL bit. L$UOSEP also accepts a 0 in R1, indicating that there are no parameters.

The **osmain** function should be declared as follows:

```
int osmain(int argc, void **argv);
```

argc is the argument count, that is, the number of pointers in the register 1 argument list. **argv** is a pointer to the unchanged list of arguments. Note that none of the arguments are tokenized, and that, in contrast to the hosted C environment, the first argument is **argv [0]**, not **argv [1]**.

The standard USS OS/390 start-up routine

The standard USS start-up routine L$UOEEP expects to be entered with R1 addressing a parameter list in the format passed by the USS **exec** system call. (See the IBM publication OS/390 UNIX System Services Assembler Callable Services, for information on this parameter list format.)

The **oemain** function should be declared as follows:

```
int oemain(int argc, char **argv);
```

argc and **argv** have the same meanings as for a regular C **main** function: **argc** is an argument count, and **argv** is a list of pointers to arguments as passed by the caller of **exec**. The first argument **argv[0]** will contain a pointer to the program name, assuming it was set properly by the caller of **exec**.

The **exec** system call passes L$UOEEP a list of environment variables specified by its caller. These environment variables can be accessed by the program using the standard **getenv** function.

The standard CMS start-up routine

The standard CMS start-up routine L$UCMSE expects to be entered with R1 and R0 set up by SVC 202 as described in VM/SP CMS for System Programming, or by SVC 204 as described in the VM/XA SP CMS Application Development Guide for CMS. In 370 mode CMS, R1 contains a code in the high-order byte and a pointer to a tokenized PLIST in the 3 low-order bytes, and R0 may contain a pointer to an extended PLIST. In XA CMS, R1 addresses a tokenized PLIST, R0 may contain a pointer to an extended PLIST, and additional information is stored at offset 96 from R13.

The **cmsmain** function should be declared as follows:

```
int cmsmain(int ecode, char *plist, char **eplist,
               char *userinfo);
```

ecode is the entry code from the high-order byte of register 1 or from the save area extension in XA CMS. **plist** is the tokenized PLIST address from register 1. **eplist** is the contents of register 0 on entry to L$UCMSE, which, depending on the value of **ecode**, may or may not address an extended PLIST. **userinfo** addresses the save area extension at 96 from R13 in XA CMS. (In 370 mode CMS, this argument is not meaningful.)

The standard CICS start-up routine

The standard CICS start-up routine L$UCICE should be entered with R1 addressing the standard CICS parameter list that contains the address of the EXEC interface block and the address of any COMMAREA, or the value X'FF000000', if no COMMAREA exists.

Declare the function **cicsmn** as follows:

```
int cicsmn (int argc, void **argv);
```

argc is set to 2. **argv** is a pointer to the unchanged list of arguments. None of the arguments is tokenized; the first argument is **argv[0]**, not **argv[1]**.

Using the indep Compiler Option with SPE

The **indep** compiler option causes the compiler to generate code so that C functions have the following special properties:

- The functions can be called before a C framework has been created.
- The functions do not expect R12 to address the CRAB at entry.
- There is no requirement on the contents of R13 except that it address a standard 72-byte save area.

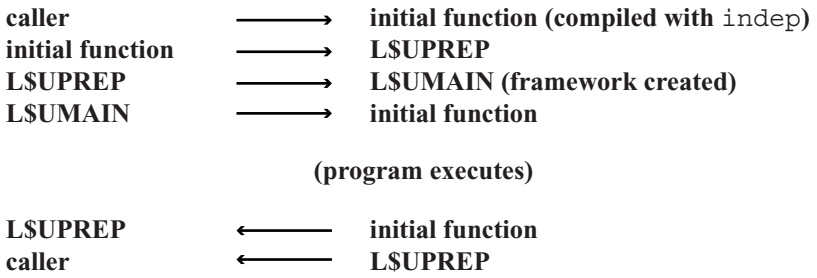
When a function compiled with the **indep** option is called, it immediately transfers control to a routine named L$UPREP. If R13 addresses a C DSA, L$UPREP takes no special action. If R13 does not address a DSA, but the framework has been created, L$UPREP loads the CRAB address into R12 and returns to the function. If the framework does not yet exist, L$UPREP calls L$UMAIN to create the framework and save the CRAB address where it can be located on future calls. After L$UPREP has completed processing, the called function resumes with the same value in R1 (and therefore the same arguments) as when it was entered.

Because any function compiled with the **indep** option can cause the framework to be created or restored, any such function can serve as a program's initial entry point or as an entry point for subsequent calls.

The **indep** compiler option can be used together with the **armode** compiler option. The result is a function that can be called before a C framework has been created and that runs in access register mode. Note that such a function must still be entered in primary address space mode. Once it completes initialization, the generated code will zero all the access registers and switch into access register mode. It is possible that an implementation of L$UPREP could support calls to an indep C function in access register mode if it was capable of saving the access registers and returning to primary address space mode before creating or restoring a SAS/C framework.

The flow of control on the first call to the program is shown in Figure 14.2 on page 285.

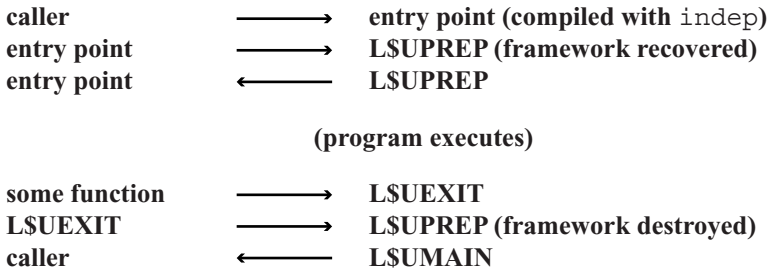
Figure 14.2 Flow Control on the First Call to the INDEP Program



With one exception, L$UMAIN executes as described in Option 2 under "The L$UMAIN Routine" on page 280. The exception is that when the SPE framework is created due to a call to a function compiled with the **indep** option, the framework is not destroyed on return from that function. This means that additional calls can be made to C functions and that external variables, memory allocated with **malloc**, and so on, will be available. To destroy the framework, you must call L$UEXIT, either directly or indirectly with **exit**. If, however, the first C function is named **main**, then the C framework is destroyed when **main** returns.

Figure 14.3 on page 285 shows the flow of control for a call when the framework already exists and when L$UEXIT is called to destroy the framework.

Figure 14.3 Flow of Control on Subsequent Calls



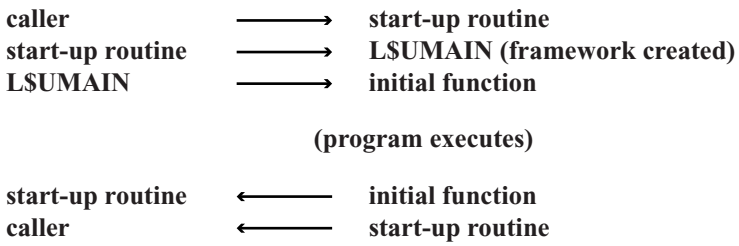
This method enables SPE programs to be invoked without a start-up routine. In general, the caller is responsible only for creating a parameter list that can be used in a C function. If this is not possible, L$UPREP can be modified to create a usable parameter list. (See "Example 2: A CMS nucleus extension start-up routine" on page 288.) L$UMAIN takes on the entire responsibility for creating the framework.

Writing a start-up routine with the indep option

Of course, frameworks can be created with both a start-up routine and one or more entry points compiled with the **indep** option. Again, this means that the framework can be retained across multiple calls to the program.

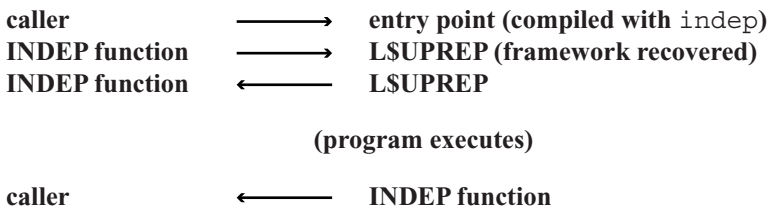
Figure 14.4 on page 286 shows the flow of control for the first call to this type of program. Because the initial function was called from L$UMAIN, L$UPREP does nothing and returns control to the initial function. Control is not returned to L$UMAIN after the initial function returns, so the framework is not destroyed.

Figure 14.4 Flow of Control in First Call to an INDEP Program Using a Start-up Routine



The flow of control for a subsequent call to this program is shown in Figure 14.5 on page 286.

Figure 14.5 Flow of Control on Subsequent Calls



The framework can be destroyed with L$UEXIT.

If you use a start-up routine, you need to be sure to store the CRAB address in a location where L$UPREP can locate it on future calls. For example, the CRAB address can be stored in the user word of an operating system control block associated with the program. When entered, L$UMAIN expects the address of this location to be in R1. If, instead, the value in R1 is 0, then L$UMAIN does not store the CRAB address. In this case, the CRAB address can be recovered from R12 by the start-up routine after L$UMAIN is complete and then stored at the appropriate location.

L$UPREP uses the L$UCENV macro to locate the CRAB address. Refer to "L$UPREP: Framework Creation and Recovery" on page 291 for information about how the L$UCENV macro is used.

Writing Your Own Start-up Routine

Writing your own start-up routine to initialize the C framework may be desirable for any of several reasons, including the following:

- If an application is invoked with nonstandard linkage (for example, unusual parameter registers), a specialized start-up routine can process or modify the argument list for easier processing from C. The sample start-up routine for an OS/390 SVC illustrates this sort of start-up routine (see "Example 1: An OS/390 SVC start-up routine" on page 287).
- If an application requires that the framework be retained after the called function returns but requires initial processing to set up an area for use by L$UCENV, this also can be handled by a specialized start-up routine. The sample start-up routine for a CMS nucleus extensionillustrates this sort of start-up routine (see "Example 2: A CMS nucleus extension start-up routine" on page 288).

To reiterate, the start-up routine calls L$UMAIN to create the framework. The framework can be destroyed on return to the start-up routine or left active until deletion by **exit** or L$UEXIT.

Example Start-up Routines

The following example start-up routines illustrate several uses for SPE and show a number of techniques for using SPE efficiently.

Example 1: An OS/390 SVC start-up routine

L$USVCE is a start-up routine that creates the C framework for an OS/390 type 3 or type 4 SVC written in the C language. To execute C code in this environment, the following problems must be solved:

- SVC routines receive parameters in registers 15 through 1 and the addresses of system data in registers 3, 4, 5, and 7. They can return data in registers 15 through 1, thus making accessing parameter and return values difficult.
- R13 cannot be used as a save area pointer for system integrity reasons. Storing blindly into the save area can cause the system to crash if R13 has been set to address critical system data.
- Solving the R13 problem requires the start-up routine to issue a GETMAIN for a usable save area. This save area must be freed before the SVC returns, but the normal L$UMAIN linkage returns control directly to the SVC caller when exit occurs.

L$USVCE solves the parameter and return value problems in the following manner. When the main C program, **svcmain**, is called, it receives a single argument (declared **void *regs [16]**) that defines a 64-byte save area in which the SVC's input registers have been saved, in the order 0 through 15. Thus, the contents on register 1 on entry can be accessed as **regs [1]**. Return values are specified by modifying the contents of this array. (Only some modifications will have any effect because the OS/390 SVC handler always restores registers 2 through 14 itself.)

L$USVCE deliberately ignores any value returned by **svcmain**. This avoids assigning a random return code when **svcmain** returns without specifying a return value.

L$USVCE does not allow the use of **exit** because the GETMAIN technique is ineffective if **exit** is used. A technique could be devised that would allow the use of **exit**, but it is easier to assume that SVC writers are disciplined enough to avoid it.

L$USVCE implements this linkage and solves the other problems as well, in the following manner:

1. L$USVCE saves registers temporarily in the RB extended save area in order to issue a GETMAIN for two save areas. The memory is obtained from a key 0 subpool to avoid integrity problems.
2. L$USVCE points register 13 to the area created by GETMAIN and copies all saved registers to the second save area. It then calls L$UMAIN, specifying that the first C routine to call is #SVCMAIN and that R1 addresses the second save area created by GETMAIN.
3. L$USVCE contains #SVCMAIN, a static routine which is defined to use the CENTRY and CEXIT macros. It points R1 to a word containing the address of the register save area and calls the user's **svcmain** function.
4. When #SVCMAIN returns, it bypasses L$UMAIN, returning directly to L$USVCE. L$USVCE stores the address of the label SVCEXIT in the R14 slot of L$UMAIN's save area. Then, L$USVCE branches to R5 to return to L$UMAIN for destruction

of the framework. After the framework has been destroyed, L$UMAIN returns to SVCEXIT rather than to L$USVCE's caller.

5 After L$UMAIN returns to the label SVCEXIT, L$USVCE copies registers possibly modified by the program out of their save area and into the RB. It then frees its save areas, reloads the necessary registers, and returns.

The techniques used by L$USVCE are applicable to a wide variety of situations in which a C program must be called with nonstandard linkage conventions.

Example 2: A CMS nucleus extension start-up routine

L$UNUXE is a start-up routine that creates the C framework for a CMS nucleus extension written in C. L$UNUXP is a specialized version of L$UPREP for the same environment. To execute C code in this environment, the following problems must be solved:

- The values in R0 and R1 on entry to the nucleus extension must be passed to the C entry point in a usable form.
- The C framework should be created only the first time the nucleus extension is called and reused on subsequent calls. The framework should be destroyed only when the nucleus extension is dropped.
- Provision should be made for a variety of nucleus extension attributes (for example, IMMCMD).

Some of these problems can be solved by using an initial function compiled with the **indep** option, but the other requirements are best solved by a specialized start-up routine and L$UPREP.

L$UNUXE solves the problems of defining the necessary nucleus extension attributes as follows. First, it presumes that the MODULE (that is, its own code) has already been defined as a nucleus extension via the CMS NUCXLOAD command.

Then, it defines a second, overriding nucleus extension with the same name, whose entry point is defined as the C function **nucxep**. This extension is always defined with the SERVICE attribute so that it gets control when the nucleus extension is dropped. The environment is not destroyed automatically when this happens. Instead, the application is expected to detect the RESET call itself (by testing for the RESET parameter) and call **exit** to destroy the framework. To allow attributes in addition to SERVICE, attribute bits from the external variable **_nucxopt** can be added to the NUCEXT argument list. (Note that the overriding nucleus extension is defined only after the C framework has been created because the values of externals are not available before this time.) The address of the CRAB is stored in the user word of the SCBLOCK associated with the nucleus extension.

L$UNUXP solves the problem of reaccessing the C framework after it has been created by the first call. The second and subsequent calls invoke **nucxep** directly, which, because it is compiled with the **indep** option, immediately calls the L$UPREP entry point of L$UNUXP. L$UNUXP retrieves the CRAB address from the user word of the SCBLOCK (addressed via register 2 on entry). Because the framework must have already been created, if this value is 0, L$UNUXP issues a diagnostic abend.

The problem of getting the R0 and R1 values to the first C function is solved jointly by L$UNUXE and L$UNUXP. In each case, the value of R1 on entry to **nucxep** is modified to address a parameter list containing the contents of R0, the original contents of R1, and the address of the save area extension. On the first call to **nucxep**, the call is made from L$UNUXE, and **nucxep** builds the new argument list. On subsequent calls, **nucxep** always calls L$UPREP immediately. L$UNUXP modifies R1 to address the R0 slot of the previous save area before returning to **nucxep**, stores the save area extension address in the R2 slot, and updates the DSAPARMS field of the DSA accordingly.

The techniques of L$UNUXE and L$UNUXP are applicable to a wide variety of situations in which the C framework must be retained for a number of separate invocations and the use of unmodified **indep** is not adequate.

SPE Internals

The previous section explained how the SPE framework is created and destroyed with start-up routines and L$UMAIN. This section discusses other SPE framework routines that are required at execution time.

The first routines discussed are the routines that handle the stack, L$UPROL and L$UEPIL. Next, the details of L$UPREP are covered. Following this discussion is an explanation of L$UTFPE, the routine responsible for handling math function exceptions. Then, L$UTZON, a routine that allows you to define the offset between local time and Greenwich time, and L$UWARN, the routine that handles conditions that call for a diagnostic message, are covered. Finally, the routine L$UHALT, which is called by the library to force abnormal termination, is described.

L$UPROL: Stack Manipulation

Implementation of C under the IBM 370 architecture requires the use of a software stack for storing **auto** variables, temporary results, and items of miscellaneous status, such as saved registers. Each function must obtain stack space on entry and free stack space on return. Stack management is provided by prolog and epilog routines. The compiler generates instructions to call these routines on function entry and return.

Because the prolog and epilog are used for every function call, their performance has a significant impact on the performance of the entire program. But there is a trade-off between the performance of the prolog and epilog and their functionality. For convenient debugging, it can be helpful to add instructions to the prolog and epilog to save additional status information, even though performance is reduced.

While the exact behavior of the prolog and epilog can be changed according to the needs of the framework, note that many components of the SPE implementation are interdependent with the prolog and epilog. Notably, L$UEXIT (**exit**), L$UJUMP (**longjmp**), L$UZSIR (USS signal handling), L$UEXLK (**bldexit**), and L$UBTRC (**btrace**) are closely involved with the details of stack management. Any change to the prolog and epilog has the possibility of some effect on these routines.

The prolog entry point is L$UPROL, which is called when a function that requires a DSA is entered. The epilog entry point is L$UEPIL, which is called when a function that requires a DSA returns. Linkage to both routines is indirect. The addresses of L$UPROL and L$UEPIL are stored in the CRAB (in fields CRABPRLG and CRABEPLG) by L$UMAIN as part of framework creation. The compiler generates code to load these addresses and branch to them. (L$UPROL is also called by L$UPREP.)

Prolog conventions

In each C function, the compiler generates code to branch to the prolog, using the following conventions:

- Registers 14 through 5 are saved in the current R13 save area. Other registers are saved only if used by the called function.
- R5 contains an address that can be used to return control to the function.
- R14 addresses the prolog entry point.

- R15 addresses the function entry point. The area immediately following this address is a control block containing such information as the size of the DSA required by the function. (This area is mapped by the CPROLOG macro.)
- R1 addresses the function's argument list.

The prolog returns to the function via R5. To return, the prolog branches to offset X'3E' (that is, the symbolic name CPROGO) from the address in R5. Usually, when the prolog is invoked, R5 and R15 have the same value, but this is not necessarily so for INDEP functions.

When the prolog returns to CPROGO, the code generated by the compiler at this point expects the following to have been done:

- R13 must address a new DSA of the requested size. The second word of the DSA must address the previous save area.
- R1 and registers 5 through 12 must be unchanged.
- R4 must address the constant CSECT, which can be loaded from the field labeled CPROCONS in CPROLOG.
- The value in R1 must have been saved in the DSAPARMS field of the new DSA.

Although not required by compiled code, the epilog requires that the value of R15 on entry be saved. L$UPROL stores this value in the DSAPRBSV field of the new DSA.

Epilog conventions

The compiler generates code to branch to the epilog, using the following conventions:

- R14 addresses the epilog entry point.
- R13 addresses the DSA for the current function.
- R1 addresses the save area for the previous function.

To return to the function, the epilog branches to offset X'36' (symbolic name CPROEXIT) from the function's entry point. Note that the epilog cannot obtain the function's entry point from the R15 slot of the previous save area because the function may have stored a return value there. R1 and registers 6 through 12 must be unchanged when the epilog returns. Registers 2 through 5 and R13 are restored as necessary by compiler-generated instructions in the function.

L$UPROL operation

The SPE version of L$UPROL uses a single block of memory allocated by L$UMAIN during framework creation as a stack and issues an abend if it overflows. This is a common way to implement a software stack, representing a compromise between maximum dependability and maximum performance.

L$UPROL performs the following steps. Certain operations may be useful for debugging and are marked as optional. In the object code for L$UPROL, optional steps have been disabled. L$UPROL

1. stores registers 6 through 11 (optional).
2. checks for stack overflow and abends if a new DSA cannot be allocated.
3. updates the stack top pointer.
4. stores the address of the previous save area in the new DSA.
5. stores the address of the new DSA in the previous save area (optional).
6. saves R15 in DSAPRBSV for the epilog.
7. copies an eye-catcher and a flag byte into the first word of the DSA. (Unless the program is in an INDEP framework, the eye-catcher is useful only for debugging.)

8 links the new DSA to the previous DSA (which may not be the same as the previous save area if there is an intervening non-C routine), and saves the new DSA address in the CRAB.

9 copies the function name to the DSAOWNER field (optional).

10 saves R1 in the DSAPARMS field.

11 loads R4 with the address of the constant CSECT from the CPROCONS field.

12 returns to the function.

L$UEPIL operation

The L$UEPIL entry point in L$UPROL performs the following steps:

1 updates the stack top and current DSA fields in the CRAB

2 loads R14 with the address of the entry point of the function from the DSAPRBSV field

3 restores registers 6 through 11 (optional)

4 returns to the returning function at the CPROEXIT offset.

L$UPREP: Framework Creation and Recovery

L$UPREP is mentioned in the preceding section as one of the INDEP framework support routines. Its function is to determine if the C framework has been created. If it has been created, then L$UPREP recovers the framework and returns to the calling function. If not, then L$UPREP calls L$UMAIN to create the framework.

L$UPREP makes a distinction between two types of function calls based on the register save area (addressed by R13) associated with the caller. The caller may be a C function, in which case the save area is part of a C DSA, or a non-C function, in which case the save area is not a C DSA. As mentioned in the L$UPROL discussion, C DSAs are distinguished by the "CSA" marker at offset 0 from R13. Note that functions written in assembler using the CENTRY and CEXIT macros are indistinguishable from functions written in the C language and are therefore considered to be C functions. Non-C functions include routines written in another high-level language, assembler routines that do not use CENTRY and CEXIT, and the operating system components.

L$UPREP is used only in an INDEP framework. When the **`indep`** compiler option is used, the code generated by the compiler to call the prolog is changed. The **`indep`** option causes the compiler to generate a branch to the L$UPREP routine. A function compiled with the **`indep`** option takes this branch almost immediately after entry.

L$UPREP takes one of the following three paths:

- If the function was not called from a C function and if the framework does not exist, it calls L$UMAIN to create the framework.
- If the function was not called from a C function, but the framework does exist, it restores the framework and returns to the called function.
- If the function was called from a C function, it returns to the called function.

When L$UPREP is entered, it inspects the save area addressed by R13 to determine whether the caller is a C function. If "CSA" appears at offset 0 (the DSACSA field in the DSA), then L$UPREP assumes that the caller is a C function.

If L$UPREP does not find "CSA", it invokes the L$UCENV macro to determine if the framework has already been created. L$UCENV returns the address of a location where a pointer to the CRAB should be (or has been) stored. If the location contains 0, then the framework has not been created. If it has been created, then L$UPREP loads the CRAB address into R12.

If the framework has been created, then, before returning control to the function, L$UPREP checks to see if the function requires a DSA. If it does, L$UPREP invokes L$UPROL to create a DSA.

When L$UPROL returns, L$UPREP marks the DSA as one belonging to a function compiled with the **indep** option and sets the DSANJUMP flag in the DSAFLGT field. This flag prohibits a **longjmp** over the function. This prohibition is established for functions compiled with the **indep** option because the caller of such a function may be written in another language, and most high-level languages do not expect **longjmp** type returns. If the caller can handle this sort of branching, the DSANJUMP flag does not need to be set.

If the framework has not been created, L$UPREP calls L$UMAIN to create it. L$UPREP loads R1 with the address returned by L$UCENV so that L$UMAIN will store the CRAB pointer for later recovery. After the framework has been created, L$UMAIN calls the function directly, placing its own return address in R5. Upon entry, the function again calls L$UPREP immediately. (Note that this is a recursive call.) Using the logic described above, L$UPREP determines that the function was called from a C function.

L$UPREP also enforces a basic convention of the INDEP framework: if the called function is the **main** function, the framework is destroyed when **main** returns by calling L$UMAIN. However, in the general case, upon return from the function, L$UPREP restores the registers (including R14) from the save area of its caller's caller and returns to the address in R14. This branch transfers control back to the routine that invoked the function that created the INDEP framework. L$UMAIN's save area, anchored in CRABMDSA, is left intact. This leaves the C framework in place for subsequent calls.

L$UPREP in the full C framework

The SPE L$UPREP is identical to the standard C library L$UPREP and can be used to replace the standard C library L$UPREP. Refer to Appendix 6, "Using the indep Option for Interlanguage Communication," on page 393 for more information.

The L$UCENV macro

When the C framework is created, L$UMAIN stores the CRAB address in some appropriate location. The L$UCENV macro defines a CSECT also named L$UCENV for this purpose. When the macro is invoked by L$UPREP, L$UCENV returns the address of the CSECT as the address of the CRAB pointer. Because this implementation forces the program to be non-reentrant, applications that need to be reentrant should use a different method of storing the CRAB address.

Under CICS, the L$UCENV macro uses the first word of the CICS transaction work area (TWA) to store the address of the CRAB pointer. You may need to decide whether or not this technique is appropriate for your application's environment. The storage allocated by the CICS SPE library is CLASS=USER; this storage is released automatically at task termination.

L$UTFPE: Math Error Handling

L$UTFPE handles floating-point error conditions such as overflow and underflow. L$UTFPE can be invoked by character to floating-point conversion functions such as **strtod** and **sscanf**. The SPE L$UTFPE uses **bldexit** and the SPIE/ESPIE SVCs to handle these conditions. Of course, other implementations may be possible that do not rely on these SVCs. The sample code for L$UTFPE is not supported in CICS.

Note that L$UTFPE uses the ESPIE macro only if the program is executing in 31-bit addressing mode. This means that the ESPIE SVC is never used under 370 mode CMS, which does not support the ESPIE SVC.

The L$UTFPE source module defines a function named L$CTFPE, which is the name of the corresponding full library implementation. The full library requires its own implementation of L$CTFPE and does not execute correctly with the SPE implementation.

L$CTFPE is called as a normal C function. It is defined as follows:

```
struct FPE {
      union {
         char space [12];
         int active;
      } hdr;
      jmp_buf get_away;
   };

   void L$CTFPE(int func, struct FPE *elt);
```

The **func** argument to L$CTFPE is either 1, to define a floating-point error trap, or 0, to cancel a previously defined trap.

When **func** is 1, **elt** addresses a trap element containing work space and a jump buffer. L$CTFPE must set **elt->hdr.active** to a non-zero value to indicate that the trap element is active. When a floating-point error occurs, the defined trap should perform the following:

```
longjmp(elt->get_away, ic)
```

ic is the program check interrupt code.

When **func** is 0, **elt** addresses the trap element for the trap to be cancelled. L$CTFPE must reset **func->hdr.active** to 0 to show that the trap has been cancelled. Note that the library routines that call L$CTFPE always cancel traps in last-in, first-out order. Also note that only one trap is ever defined at a time unless a function such as a math function, is interrupted by a user **bldexit** routine that also calls a math function.

L$UTZON: Local Time Offset Determination

L$UTZON is called by library timing functions to obtain the difference between Greenwich time and local time. The timing routines assume that **time_t** values contain Greenwich time and use the information returned by L$UTZON to convert them to local time. L$UTZON supports several return value formats, since some information is more readily available in some environments than in others.

The L$UTZON routine is also used in the SAS/C Generalized Operating System (GOS) interface. The linkage conventions for L$UTZON in SPE and GOS are similar enough that the same routine can be used for both. See SAS Technical Report C-115, The Generalized Operating System Interface for the SAS/C Compiler Run-Time System, Release 5.50 for more information.

When L$UTZON is called, register 1 addresses a parameter list in the format shown in Example Code 14.1 on page 293.

Example Code 14.1 L$UTZON Parameter List Format

```
TZONPRMS DS   0D
         DS   A       zero (nonzero for GOS)
         DS   A       zero (can be used as a work area)
```

```
         DS   A       address of a doubleword return value
```

Register 13 addresses a standard save area when L$UTZON is called; however, it is not necessary to save and restore registers, as this is done by the caller of L$UTZON.

When L$UTZON returns, the value in register 15 indicates the format and meaning of the data addressed by the third word of the parameter list.

If L$UTZON returns a code of 0, it stores a signed integer in the first word of the return area, indicating the number of seconds difference between local time and Greenwich time. For example, if it is 4 p.m. locally when it is 2 p.m. Greenwich time, +7200 (2 hours in seconds) is stored.

If L$UTZON returns a code of 1, it stores a value in TOD clock format in the return area, indicating the local time. More precisely, this value represents the number of seconds since the local midnight of January 1, 1900, where bit 51 of the doubleword represents a microsecond.

If L$UTZON returns a code of 2, it stores the local date and time in the return area in the format used by the OS TIME BIN macro. More precisely, the first word of the doubleword should contain the local time, expressed as the number of hundredths of a second since midnight, represented as an unsigned binary integer. The second word of the doubleword should contain the packed decimal local date in the form 00YYDDDF, where YY is the number of years since 1900, and DDD is the number of days since January 1.

If L$UTZON cannot determine the local time offset, it should return a code of 8 in register 15.

L$UWARN: Issue Diagnostic Messages

Some SPE library functions, such as **memcpy** and **sqrt**, are designed to issue diagnostic messages. In the SPE framework, the routine L$UWARN is called whenever a diagnostic is appropriate. The SPE version of this routine simply stores an appropriate value in **errno** and returns. Depending on the needs of the application, some other action, such as issuing an abend or actually writing a diagnostic, may be preferred.

L$UWARN can be called through either of its entry points, #WARNING or $WARNING. When it is called, R1 addresses a variable length parameter list in the format shown in Example Code 14.2 on page 294.

Example Code 14.2 L$UWARN Parameter List Format

```
WARNPRMS DS   0D
         DS   F       message number
         DS   F       value to be stored in errno
         EQU  *       zero or more replacement values
         .
         .
         .
```

The first two words in the list are the diagnostic message number and the value to be stored in **errno**. Any additional arguments represent values to be inserted into the message text. (These values can be processed using the **va_arg** macro.)

Two special **errno** values should be noted. If the value to be stored in **errno** is 0, the diagnostic is a note rather than a warning, and **errno** should not be changed. If the value to be stored is negative, it indicates a severe error, and an abend is recommended.

If L$UWARN is to write diagnostic messages, obtain the message texts from the SASC.ERRMSGS data set (under OS/390) or LSU ERRMSGS (under CMS). Each record in this file contains a message number in columns 1-8 and the corresponding message

text beginning in column 9. The message texts are suitable for use as formats with the **vsprintf** function.

L$UHALT: Terminate Execution Abnormally

After certain error conditions, the library needs to abnormally terminate program execution. For instance, if the program calls the POSIX **getpid** function, but USS is not running, execution cannot continue because the function call cannot succeed. However, the function definition does not provide a way for the function to fail. The SPE library forces abnormal termination by calling the routine L$UHALT. The supplied version of this routine simply issues the assembler ABORT macro, which, in all systems other than CICS, issues an ABEND.

L$UHALT is called via the entry point L$CHALT. When it is called, R1 addresses a parameter list in the format shown in Example Code 14.3 on page 295.

Example Code 14.3 L$UHALT Parameter List Format

```
HALTPRMS DS   0D
         DS   F          intended ABEND code
         DS   F          message suppression flag
```

The first word in the argument list is the intended ABEND code, an integer between 1200 and 1240. The second argument is an integer which, if not zero, requests suppression of any library messages about the ABEND. Since the SPE library does not diagnose ABENDs, this argument can be ignored.

Note that if L$UHALT returns to its caller, the effects of further execution are completely undefined.

Interrupt Handling in SPE

A frequent requirement for systems programming applications is the need to write synchronous or asynchronous exits, such as SPIE or STIMER exits. A C program cannot merely issue the appropriate SVC to define an exit, as an assembler language program does, for the following two reasons:

- The exit routines use a wide variety of inconvenient, inconsistent, and incompatible linkage conventions.
- The operating system does not provide the environment expected by a C function to an exit routine.

For example, consider an application that wants to use a C function as a SPIE exit. The function cannot be called directly by the operating system because R13 will not address a usable save area and R1 will not address a C format argument list. Even if it were callable, the C function would not be able to share any data with its caller.

However, it is possible for the exit to call a block of code that establishes (or re-establishes) the C framework and then transfers control to a C function. The code, of course, must be tailored to the specific exit so that the exit's linkage conventions can be honored. In addition, the code has to handle the transfer of control back from the function to its caller. Ideally, the code does no more than necessary to transfer control and does not make any assumptions about the way the calling exit and the called C function transfer data.

In SPE, the **bldexit**, **freeexit**, and **bldretry** functions provide this service. These functions can be used to build exit linkage code that can mediate between the operating

system and the C function. (These functions are exclusive to SPE. The standard signal handling functions provide similar services, portably, in the full C framework.)

The bldexit Function

The **bldexit** function creates a sequence of instructions that establishes linkage between an operating system exit and a C function. **bldexit** takes two arguments, a pointer to the C function that is to be called and a flag word that describes the required linkage. (The flag word is described in detail in the **bldexit** function description later in this chapter.) **bldexit** returns the address of the linkage code, which can then be passed to the SVC that establishes the exit. For instance, suppose a function **tmrexit** is to be called from an STIMER exit. The following statements call **bldexit** to create the appropriate linkage and then invoke STIMER:

```
unsigned intvl = 1000;   /* ten second time interval */
void *exit_addr;         /* pointer to linkage code */

exit_addr = bldexit(&tmrexit, _ASYNCH+_NOR13);

   /* Set up registers for STIMER SVC. */
_ldregs(R0+R1+R15, 0x90000000, &intvl, exit_addr);

   /* Issue STIMER. */
_ossvc(47);
```

When the timer interrupt occurs, the operating system calls the linkage code built by **bldexit**. This code saves registers as necessary, re-enters the C framework, and calls **tmrexit**. When **tmrexit** returns, the exit linkage code returns control to the operating system. An exit function called by **bldexit** has the following general definition:

```
void exit_fun(void **sa, char **poi);
```

The **sa** argument addresses a save area where the linkage code saves the contents of all the general registers on entry, in the order 14 through 12. (For example, the contents of register 1 on entry are accessed as **sa[3]**.) This allows the exit function access to all data passed to the exit. The exit routine can return data to the operating system in any register by modifying the corresponding word in the save area. For example, it can use the following assignment to return 4 in register 15:

```
sa [1] = (void *) 4;
```

The **poi** argument addresses a fullword where the exit can store the point of interrupt (such as an old PSW), if this is meaningful. This information is used by the **btrace** library function to produce a correct backtrace in the presence of interrupts. There is no reason to store a point of interrupt if you do not call **btrace**. In some cases, you may not be able to determine a point of interrupt. You can still call **btrace** in this situation, but the resulting output may be incomplete.

The bldretry Function

Some system exit interfaces, such as SPIE and ESTAE, allow the assembler programmer to request a retry, which causes program execution to resume at a point other than the point of interruption. The SPIE SVC requires the exit to request a retry. Just as defining a C function as an exit does not work, using a C label as a retry address also does not work. **bldretry** is an interface similar to **bldexit**, with some differences due to the special requirements for retry routines.

Just as **bldexit** serves as a mediator between exit linkage and the C function call mechanism, **bldretry** serves as a mediator between retry linkage and the C **longjmp** interface. You can think of **bldretry** as a method of issuing **longjmp** from an exit function. Of course, the exit must support a retry interface for this to be effective. **bldretry** is passed two arguments, a **jmp_buf** defining the retry location and an integer jump code. **bldretry** builds linkage code for the retry and then returns the address of this code, which can be passed to the operating system to perform the retry.

Example Code 14.4 on page 297 shows two code fragments that define a retry location using **setjmp** and request a retry from an ESTAE exit routine.

Example Code 14.4 ESTAE Retry Using setjmp and bldretry

```
    /* 1. Define post-ABEND retry point. */

if (code = setjmp(ESTAE_jmp_buf)) {
    /* If ABENDed, retry here. */
}
    /* normal execution path */


    /* 2. Request retry within the ESTAE exit routine. */
SDWA->SDWARTYA = bldretry(ESTAE_jmp_buf, 1);

    /* Store retry address in SDWA. */
SDWA->SDWARCDE = SDWARETY;
    /* Tell ABEND to retry. */
return;
```

In the example, after the **return** is executed, the retry linkage code is entered. This code performs the equivalent of the following:

```
longjmp(ESTAE_jmp_buf, 1)
```

It returns control to the C program at the point where the **setjmp** function is called.

Unlike **bldexit** linkage code, **bldretry** linkage code can be used only once. The code is freed before control returns to the C program.

The freeexit Function

The **freeexit** function frees the memory used for the linkage code created by **bldexit**. Obviously, **freeexit** should not be called until the corresponding exit routine is no longer defined to the operating system.

A complete example of the use of **bldexit** and **bldretry** can be found in the source code for the L$UTFPE module. This routine uses the SPIE and ESPIE SVCs to handle computational program checks for the library math functions.

Issuing CICS commands

You can use EXEC CICS commands in SPE applications written in C in exactly the same way as with the full library. Refer to Chapter 2, "The SAS/C CICS Command Preprocessor," in the SAS/C CICS User's Guide for details on using the SAS/C CICS command preprocessor. The use of the EXEC CICS RETURN or EXEC CICS XCTL

command in an SPE application terminates the C environment before execution of the command. Registered **atexit** routines are also called before execution of the command.

Note the following:

- The HANDLE command is not supported in C code by SPE.
- The **_eibptr** global variable is set in L$UMAIN to point to the EXEC Interface Block. The global externs **_commptr** and **_dibptr** are not initialized by any SPE code. No DL/I initialization call is performed.

In addition to using CICS commands in your SPE C code, you may find it necessary to add CICS commands to library SPE routines in assembler when you modify them. EXEC CICS commands in assembler programs are translated into invocations of the DFHECALL macro, which uses EXEC interface storage, mapped by DFHEISTG, to build parameter lists to pass to CICS. The initial allocation of this storage occurs in L$UMAIN; its pointer is stored in the CRAB control block field CRABEIS. You can address this storage in the following manner:

```
L     Rx,CRABEIS
USING DFHEISTG,Rx
EXEC  CICS ....
```

Alternatively, you can provide a storage area in the DSA of an assembler function for the DFHECALL macro's use, as in the following example:

```
COPY DSA
DFHEIPL   DS   20F
DSALEN    EQU  *-DSA
```

Writing CICS User Exits

The sample source code issues several EXEC CICS commands:

- GETMAIN
- FREEMAIN
- ADDRESS EIB
- LOAD
- RELEASE

If you want to write CICS exit programs in C, you must modify these commands. For example, in CICS/ESA, code the corresponding user exit programming interface (XPI) calls rather than issue EXEC CICS commands. You can modify L$UCENV to save the CRAB address in a global work area so that the C framework is only created when the program is first invoked; subsequent invocations restore the C framework.

SPE and USS

This section provides miscellaneous information about systems programming under USS. Refer to SAS/C Library Reference, Volume 2 for additional information about USS and SAS/C POSIX support.

USS Interface

All uses of USS system calls in SPE are routed through a single function, L$CUBPX, with the exception of a few signal-handling calls. In addition to issuing the requested

system call, L$CUBPX is responsible for translating USS error codes into SAS/C error numbers and for detecting failures in USS itself. L$CUBPX is provided in source (member L$UUBPX in the SPE source library) to allow it to be tailored if necessary.

Timing Functions

The SPE library does not support POSIX time zones. Time zone information is determined exclusively by the SPE L$UTZON routine.

The support for the **_epoch** external variable is available in SPE as well as in the regular library. Note that the SPE default epoch is the UNIX epoch beginning January 1, 1970. This is the same default as for the regular library.

HFS Access

SPE supports access to Hierarchical File System (HFS) files using the low-level routines such as **open**, **read**, **write**, **lseek**, and so on. Note that when you use SPE, these functions can only access the hierarchical file system; that is, you cannot use these functions to access OS/390 data sets or sockets. Filenames passed to **open** and other POSIX functions should always specify a POSIX filename. Style prefixes will be treated as part of the filename, and a leading `"//"` prefix will be treated as if it were a single slash.

When an USS SPE application receives control from the **exec** system call, file descriptors 0, 1, and 2 are normally passed by the caller of **exec**. The SPE library does not use or require these file descriptors, but they are available for the use of the application.

Note that HFS I/O can only be performed using low-level POSIX functions, as the standard I/O functions are not supported in SPE. The POSIX functions **fdopen** and **fileno** are also not available, since they are used in the context of standard I/O.

SPE supports access to USS integrated sockets using the standard UNIX socket interface functions such as **socket**, **accept**, **read**, **write**, and so on. Note that only integrated sockets can be accessed. Also, TCP/IP configuration information functions, such as **gethostbyname** and **getservent**, are not supported because USS does not provide system calls for these functions.

Environment Variables

Environment variables are now supported by SPE for both USS and other applications. However, the only way to create an environment variable in SPE is to call **putenv** or **setenv**. That is, there is no library processing to copy environment variables from any external source. Note that for programs called via the **exec** system call, the L$UOEEP start-up routine is responsible for setting up the environment variables passed by **exec**.

Environment variable names are always case sensitive when SPE is used.

Signal Handling

SPE supports USS signal handling. Only signals supported by USS can be handled. (For instance, the SAS/C signal SIGMEM is not supported in an SPE environment.) Further, only functions defined by the POSIX Standard are supported. For instance, you can use the **sigaction**, **kill**, and **sigprocmask** functions in an SPE application but not the **signal**, **raise**, and **sigsetmask** functions, which are defined by ANSI or by SAS/C, not by POSIX.

Note that the timing of signal delivery is different under SPE than with the library. When an USS signal is delivered to an SPE program, the signal handler is invoked immediately, in contrast to the library case, where the signal is delayed until the signal can be discovered. Because signal handlers can be called at any time, you may need to block signals during critical sections to prevent interruptions at inconvenient times.

Note that when you use SPE, the signals SIGILL, SIGSEGV, SIGFPE, SIGABRT, and SIGABND are not by default associated with program checks and ABENDs, as in the standard library. Your application will receive one of these signals only if sent by some program using the **kill** function. If you want an SPE application to handle program checks or ABENDs, you can use the SPE-only function **oeabntrap**, which issues an **ESTAE** macro to define an exit that transforms any recoverable ABEND into an appropriate USS signal. **oeabntrap** also enables an interface to the USS **ptrace** system call, which allows an SPE program to be debugged (as an assembler program) by the IBM **dbx** debugger.

A number of SPE signal-handling routines are supplied in source to support user enhancements. The source modules are

L$UZABN	**oeabntrap** interface
L$UZEST	**oeabntrap** ESTAE exit and retry routine
L$UZOEI	initialize and terminate USS signal handling
L$UZSIA	define a signal handler
L$UZSIR	invoke a signal handler on receipt of an USS signal

L$UZSIR contains both a signal interface routine (SIR), which is called by USS when a signal occurs, and L$UZRTE, a routine which is given control by the SIR using **bldexit** linkage in order to invoke the user's handler.

fork Function

SPE supports the use of the **fork** function. The **atfork** SAS/C extension is not supported. However, a CRAB field CRABFKCT is defined to allow the application to be aware of the use of **fork**. CRABFKCT is initialized to zero. Whenever a fork occurs, it is incremented by one in the child process. This allows a function to determine whether the process id has changed as the result of **fork**. Such checks may be needed due to the fact that many OS/390 resources (for example, timers) are not copied to a child process.

POSIX Compiler Option

Whether or not the POSIX compiler option is used has no effect at runtime on an SPE application. It still has its other compile-time effects, such as defining **_SASC_POSIX_SOURCE** and implying the **refdef** option.

The SPE Library

SPE is a minimal C environment. Therefore, the SPE library is a subset of the library available in the full C library. Many of the SPE library functions are provided both in source and object code formats. The following functions comprise the SPE library and may be used in an SPE program. Functions that are not included in this list cannot be used in an SPE environment.

Note: A * after the function name indicates that the function is only supported under USS; a + after the function name indicates that a special SPE version is supplied; and a # after the function name indicates that the function is only usable with SPE. An at sign (@) after the function name indicates that the function will accept near or far pointers if compiled with the **`armode`** option. △

Table 14.1

ABEND	abs	accept *	accept_and_recv *
_access *	acos	alarm *	alarmd *
slrdrtv#	asctime	asin	atan
atan2	atexit +	atof	atoi
atol	atoll	ATTACH	bind *
bldexit #	bldretry #	blkjmp +	bsearch
btrace +	calloc	ceil	cfgetispeed *
cfgetospeed *	cfsetispeed *	cfsetospeed *	CHAP
chaudit*	chdir *	chmod *	chown *
chpriority *	clearenv +	close *	_close *
closedir *	cmsclose	cmsdfind	cmsdnext
cmserase	cmsopen	cmspoint	cmspush
cmsqueue	cmsread	cmsshv	cmsstack
cmsstate	cmswrite	cmsxflpt	cmsxflrd
cmsxflst	cmsxflwr	CMSSTOR_OBT	CMSSTOR_REL
connect *	cos	cosh	creat *
DEQ	DETACH	difftime	div
DMSFREE	DMSFREE_V	DMSFRET	DOM
DOM_TOK	dspserv #	dup *	dup2 *
ecbsuspend *	endgrent *	endpwent *	ENQ
erf	erfc	ESTAE #	ESTAE_CANCEL #
execl *	execle *	execlp *	execv *
execve *	execvp *	exit +	_exit *
exp	extlink *	fabs	falloc #
fchaudit *	fchmod *	fchown *	fcntl *
_fcntl *	ffree #	floor	fmax
fmin	fmod	fork *	format
fpathconf *	free +	freeexit #	FREEMAIN
frexp	fstat *	fsync *	_fsync *
ftruncate *	getclientpid *	getcwd *	getegid *
getenv +	geteuid *	getgid *	getgrent *

getgrgid *
getgrnam *
getgroups *
getgroupsbyname *
gethostid *
gethostname *
getitimer *
getlogin *
GETMAIN_C
GETMAIN_U
getpeername *
getpgid *
getpgrp *
getpid *
getppid *
getpriority *
getpwent *
getpwnam *
getpwuid *
getrlimit *
getrusage *
getsid *
getsockname *
getsockopt *
getuid *
getwd *
givesocket_pid *
gmtime
htoncs
hypot
initgroups *
isalnum
isalpha
isascii
isatty *
iscntrl
iscsym
iscsymf
isdigit
isebcdic
isgraph
islower
isnotconst
isnumconst
isprint
ispunct
isspace
isstrconst
isunresolved
isupper
isxdigitv
j0
j1
jn
kill *
labs
lchown *
ldexp
_ldexp
ldiv
link
listen *
llabs
lldiv
llmaxllmin
loadm +
localtim
log
log10
longjmp +
lseek *
_lseek *lstat *
malloc +
_matherr
max
mblen
mbstowcs
mbtowc
memchr
memcmp
memcmpp
memcpy
memcpyp
memfil
memscan
memscntb
memset
memupr
memxlt
min
mkdir *
mkfifo *
mknod *
mktime
mmap *
modf
mount *
mprotect *
msgctl *
msgget *
msgrcv *
msgsnd *
msgxrcv *
msync *
munmap *
ntohcs
oeabntrap #
oeattach *
oeattache *
oetaskctl *
offsetof
onjmp
onjmp
open *
_open *
opendir *
osbclose
osbdcb
osbldl
osbopen
osbopenj
oscheck
osclose
osdcb
osdynalloc
osfeov
osfind
osfindc
osflush
osget
osnote
osopen
osopenj
ospoint
osput
osread
osseek
osstow
ostclose
ostell
oswrite
palloc
__passwd *
pathconf *
pause *
pdel
pdset
pdval
pfree
pfscttl *
pipe *
pool

POST	pow	putenv +	qsort
rand	RDTERM	read *	_read *
readextlink *	readv *	readdir *	readlink *
realpath *	recv *	recvfrom *	recvmsg *
rename *	_rename *	rewinddir *	rmdir *
select *	selectecb *	semctl *	semget *
semop *	send *	sendmsg *	sendto *
setegid *	setenv +	seteuid *	setgid *
setgrent *	setgroups *	setitimer *	setjmp +
setpgid *	setpriority *	setpwent *	setregid *
setreuid *	setrlimit *	setsid *	setsockopt *
setuid *	SETRP_COMPCOD #	SETRP_DUMP #	SETRP_REASON #
SETRP_RETRY #	shmat *	shmctl *	shmdt *
shmget *	shutdown *	sigaction *	sigaddset *
sigblkjmp *+	sigdelset *	sigemptyset *	sigfillset *
sigismember *	siglongjmp *	sigpending *	sigprocmask *
sigsetjmp *	sigsuspend *	sin	sinh
sleep *	sleepd *	snprintf	socket *
socketpair *	spawn *	spawnp *	sprintf
sqrt	srand	sscanf	stat *
STATUS	stcpm	stcpma	STIMER #
STIMERM #	STIMER_CANCEL #	STIMERM_SET #	STIMERM_TEST #
strcat	strchr	strcmp	strcpy
strcspn	strerror	strlen	strftime
strlwr	strncat	strncmp	strncpy
strpbrkstrrchr	strrcspn	strrspn	strsave
strscan	strscntb	strspn	strstr
strtod	strtok	strt ol	strtoll
strttoul	strttoull	strupr	strxlt
symlink *	sysconf *	takesocket_pid *	tan
tanh	tcdrain *	tcflow *	tcflush *
tcgetattr *	tcgetpgrp *	tcgetsid *	tcsendbreak *
tcsetattr *	tcsetpgrp *	TGET	time
times *	toebcdic	tolower	toupper
TPUT	TPUT_ASID	TPUT_USERID	truncate *
TTIMER #	ttyname *	typlin	umask *
umount *	uname *	unlink *	_unlink *
unloadm +	utime *	va_arg	va_end

va_start	vformat	vsnprintf	vsprintf
w_getmntent *	w_getpsent *	w_ioctl*	w_statfs *
wait *	WAIT1	WAITM	waitpid *
waitrd	WAITT	wcstombs	wctomb
write *	_write *	writev *	WRTERM
WTO	WTOR	xedpoint	xedread
xedstate	xedwrite	xltable	y0
y1	yn		

The SPE library contains functions in the following four categories:

1. functions that have no full library equivalent and can be used exclusively in SPE (for example, **bldexit**)
2. functions that mimic full library functions but have been designed for SPE applications (for example, **malloc** and **free**)
3. functions that invoke commonly used SVCs or operating system functions (for example, **waitrd** and **FREEMAIN**)
4. functions that can be used in both the full library and in SPE.

Most of the functions that comprise the SPE library belong to category 4 and are also considered part of the full SAS/C library. Table 14.2 on page 304 lists the functions in the other three categories. The Implementation (Source) column shows whether the function is implemented as a function or as a macro and the name of the corresponding source file, if any. The functions are divided into groups according to the area of the SPE library to which they belong. The functions in category 3 would not normally be used under CICS.

Following Table 14.2 on page 304 are descriptions of the functions in categories 1, 2, and 3. The **format** function and **vformat** functions (category 4) are also documented in this section. Although these functions can be used in the full C library, **sprintf** or **vsprintf** is usually more useful in that framework.

Table 14.2 SPE Library Functions

Function	Category	Implementation (Source)
Memory Management		
aleserv	1	macro (**<osdspc.h>**)
CMSSTOR_OBT	3	macro (**<cmastor.h>**)
CMSSTOR_REL	3	macro (**<cmastor.h>**)
DMSFREE	3	macro (**<dmsfree.h>**)
DMSFREE_V	3	macro (**<dmsfree.h>**)
dspserv	1	macro (**<osdspc.h>**)
falloc	1	macro (**<osdspc.h>**)
ffree	1	macro (**<osdspc.h>**)
free	2	function (L$UHEAP)
GETMAIN_C	3	macro (**<getmain.h>**)

Function	Category	Implementation (Source)
GETMAIN_U	3	macro (**<getmain.h>**)
GETMAIN_V	3	macro (**<getmain.h>**)
FREEMAIN	3	macro (**<getmain.h>**)
malloc	2	macro (L$UHEAP)
Program Control		
atexit	2	function (L$UATEX)
exit	2	function (L$UMAIN,L$UEXIT)
oeabntrap	1	function (L$UZABN,L$UZEST)
Diagnostic Control		
btrace	2	function (L$UBTRC)
Dynamic Loading		
loadm	2	function (L$ULDR)
unloadm	2	function (L$ULDR)
Signal Handling		
bldexit	1	function (L$UEXLK)
bldretry	1	function (L$URETR)
freeexit	1	function (L$UEXLK)
sigaction	2	function (L$UZOEI, L$UZSIA, L$UZSIR)
Environmental Variables		
getenv	2	function
putenv	2	function
setenv	2	function
Terminal I/O		
RDTERM	3	macro (**<wrterm.h>**)
TPUT	3	macro (**<tput.h>**, L$UTPIO)
TPUT_ASID	3	macro (**<tput.h>**, L$UTPIO)
TPUT_USERID	3	macro (**<tput.h>**, L$UTPIO)
TGET	3	macro (**<tput.h>**, L$UTPIO)
typlin	3	macro (**<wrterm.h>**)
waitrd	3	macro (**<wrterm.h>**)
WAITT	3	macro (**<wrterm.h>**)
WRTERM	3	macro (**<wrterm.h>**)
Other System Interface		
CMS low-level I/O	4	function (L$UCMIO)
OS/390 dynamic allocation	4	function (L$UDYNA)

Function	Category	Implementation (Source)
OS/390 low-level I/O	4	function (L$UOSIO, L$UBSAM, L$UDCB)
OS/390 low-level multitasking	4	function (L$UATTA, L$UAEOT)
POSIX system calls	4	function (L$UUBPX)
General Utility		
format	4	function

The SPE library functions are described in detail throughout the remainder of this chapter in alphabetical order.

aleserv

Access list management services

SYNOPSIS

```
#include <osdspc.h>
int aleserv(char *request, ...);
```

DESCRIPTION

The **aleserv** function implements the functionality of the OS/390 assembler ALESERV macro. The **request** argument is a null-terminated string. The request must be one of the following:

"ADD"	adds an entry to the access list and return an ALET.
"ADDPASN"	adds the primary address space to the DU-AL.
"DELETE"	deletes an entry from the DU-AL or PASN-AL.
"EXTRACT"	returns the STOKEN associated with a given ALET.
"SEARCH"	returns the ALET associated with a given STOKEN.
"EXTRACTH"	returns the STOKEN of the HOME address space.

The remainder of the argument list is a list of keywords followed, in most cases, by an argument specifying a value for the keyword. The list is terminated by the **_Lend** keyword. The supported keywords and their associated data are as follows:

- The **_Lreason** keyword is used to pass back a reason code in the event that the service fails. The next argument should be a pointer to an **int** that will contain the reason code as set in R0 by the ALESERV macro if the ALESERV macro returns nonzero in R15.
- The **_Lstoken** keyword is equivalent to the Assembler STOKEN keyword. The next argument should be the address of an 8-character array. This array is filled in by the EXTRACT and EXTRACTH service and is required as input to the ADD and SEARCH services.
- The **_Laccess** keyword is equivalent to the Assembler ACCESS keyword. The next argument should be one of the following strings:

  "PUBLIC"
  The access list entry being added is public.

"PRIVATE"
The access list entry being added is private.

- The **_Lal** keyword is equivalent to the Assembler **AL** keyword. The next argument should be one of the following strings:

  "WORKUNIT"
  The access list being referenced is a DU-AL.

  "PASN"
  The access list being referenced is a PASN-AL.

- The **_Lalet** keyword is equivalent to the Assembler **ALET** keyword. The next argument should be a pointer to a 4 byte field containing an ALET that you provide, or that the system returns.

  For ADD and ADDPASN, the system returns the **ALET** of the added entry.

  For the DELETE request, you provide the **ALET** of the access list entry to be deleted. Do not specify an ALET of 0, 1, or 2.

  For the EXTRACT request, you provide the **ALET** whose STOKEN you require.

  For the SEARCH request, you specify where in the access list the system is to begin the search.

- The **_Lchkpt** keyword is equivalent to the Assembler **CHKPT** keyword. The next argument should be one of the following strings:

  "FAIL"
  If a CHKPT macro is issued, reject it.

  "IGNORE"
  The system processes the CHKPT, but the user must ensure that the dataspace, ALETs, STOKENs, and so on are restored before a restart is attempted.

- The **_Lend** keyword indicates the end of the list of keywords.

The following parms can be used by authorized programs only:

- The **_Lchkeax** keyword is equivalent to the Assembler DREF keyword. The next argument should be one of the following strings:

  "YES"
  Check the EAX authority of caller to the dataspace to be added to or deleted from the access list.

  "NO"
  Do not check the EAX authority of caller to the dataspace to be added to or deleted from the access list.

RETURN VALUE

aleserv returns 0 if the ALESERV macro was successful. If the ALESERV macro fails, it returns the return code from the macro, which will be a positive value. **aleserv** may also return -1 to indicate an unknown or invalid keyword combination.

IMPLEMENTATION

The **aleserv** function is implemented by the source module L$UDSPC.

EXAMPLE

For an example of the **aleserv** function, see the examples under **dspserv**.

SEE ALSO

dspserv, **falloc**, **ffree**

---

atexit

Register Program Cleanup Function in SPE

SYNOPSIS

```
#include <stdlib.h>

int atexit(_remote void (*func)());
```

DESCRIPTION

Refer to Chapter 6, "Function Descriptions," in SAS/C Library Reference, Volume 1 for a description of **atexit**.

RETURN VALUE

atexit returns 0 if successful or a non-zero value if unsuccessful.

PORTABILITY

atexit is portable.

IMPLEMENTATION

The SPE implementation of **atexit** is in L$UATEX. This version enforces the ANSI Standard limit of 32 registered functions.

SEE ALSO

exit

---

bldexit

Build System Exit Linkage in SPE

SYNOPSIS

```
#include <bldexit.h>

void *bldexit(__remote void (*func)(), unsigned flags);
```

DESCRIPTION

bldexit builds linkage code enabling a C function to be invoked as a synchronous or asynchronous exit routine. **bldexit** is supported only when the SPE framework is used; it is not supported with the standard C framework.

The **func** argument is the address of the C function that is called by the operating system as an exit routine. The **flags** argument is the sum of zero or more flags indicating attributes of the required exit routine linkage. Note that the exit is expected to be entered with register 14 containing a return address and register 15 serving as a base register.

The following exit attributes can be set:

_ASYNCH
: specifies that the exit can be entered asynchronously. This flag should be set if the exit can be entered while a non-C or system routine is running. This flag also should be set if the exit can be entered while a C function without a DSA (a leaf routine) is running or while the C prolog or epilog is running. Failure to specify ASYNCH when appropriate may lead to abends or stack overlays when the exit linkage code is called.

_NOR13
: specifies that the exit can be entered with register 13 addressing an area that cannot be used as a save area.

_FLOATSV
: specifies that the exit linkage must save and restore floating-point registers. FLOATSV needs to be specified only if the operating system's linkage does not save and restore floating-point registers and if the exit function (or any function it calls) uses the floating-point registers.

_AMODE
: specifies that the exit can be entered in a different addressing mode from the interrupted program and that the exit linkage must restore the original mode. This attribute is ignored in a system that does not support 31-bit addressing.

_ACCSV
: specifies that the exit linkage must save and restore access registers. _ACCSV needs be specified only if the operating system's linkage does not save and restore access registers, and if the **exit** function (or any function it calls) is compiled with the **armode** option or uses the access registers in some other way.

Note: The old exit attribute names, **ASYNCH**, **NOR13**, **FLOATSV**, and **AMODE** still work if your program does not include the header file **spetask.h** as well as **bldexit.h**. △

The linkage code generated by **bldexit** is stored in an exit element allocated with GETMAIN or DMSFREE. When the exit is no longer required, you should call the **freeexit** routine to release this storage.

RETURN VALUE

bldexit returns the address of the exit linkage code. This address should be passed as an argument to the system call that defines the operating system exit.

CAUTIONS

The **longjmp** and **exit** functions cannot be called from a routine entered via **bldexit** linkage code. An attempt to do so results in a user ABEND 1224.

Routines entered via **bldexit** must be executed as interrupts and cannot execute in parallel with the interrupted code. For example, **bldexit** cannot be used for code that runs under an SRB because if the code takes a page fault, the interrupted C code can resume execution, causing stack overlays and other disasters.

PORTABILITY

bldexit is not portable.

USAGE NOTES

See "Interrupt Handling in SPE" on page 295 for argument specifications for **bldexit** exit routines.

Source for **bldexit** is supplied in L$UEXLK. You can modify this source to define attribute bits to support exits with unusual linkages not already supported.

EXAMPLE

This example shows use of **bldexit** with the SPIE SVC to define a C function to be called if a protection or addressing exception occurs. The C function writes a backtrace using the **btrace** function and executes a retry in order to terminate the program's execution with a failure return code.

```
#include <osmain.h>
#include <setjmp.h>
#include <bldexit.h>
#include <svc.h>
struct PICA {  /* Map the Program Interrupt Control Area. */
   char *exit;
   unsigned short bits;
};
#define SPIE(pica, addr, mask) \
        (pica.bits = mask, pica.exit = addr, \
        *(char *) &pica = 0x0f,_ldregs(R1, &pica), \
        _ossvc(14), _stregs(R1))
int oldpica;       /* previous program's PICA */
jmp_buf retrybuf;
static void pgmchk();
extern void msgwtr();  /* unshown message writer routine */
osmain()
   {
      struct PICA my_PICA;
      void *exitloc;
      int rc = 0;     /* success or failure code */
      if (setjmp(retrybuf)) goto pgm_check;
      exitloc = bldexit(&pgmchk, _ASYNCH+_NOR13);
      oldpica = SPIE(my_PICA, exitloc, 0x0c00);
         /* Intercept 0C4 and 0C5, */
         /* then do some real work. */
      goto quit;
pgm_check:
      rc = 16;       /* Set program failure code. */
quit:
      _ldregs(R1, oldpica);
      _ossvc(14);    /* restore old SPIE */
      freeexit(exitloc);
      return rc;
   }
static void pgmchk(sa, poi)
   void **sa;
   char **poi;
   {
      struct {       /* the Program Interrupt Element */
         struct PICA *pica;
         short misc1;
```

```
        short int_code;    /* interrupt code              */
        char *addr;        /* location of interrupt/retry */
        char *regs[5];     /* saved registers             */
    } *PIE;
    char msgbuf[40];
    PIE = sa[3];           /* R1 addresses PIE on entry     */
    *poi = PIE->addr;      /* store program check location  */
                           /* for btrace                    */
    format(msgbuf, "Program check %d!", PIE->int_code);
    msgwtr(msgbuf);
    btrace(msgwtr);
    PIE->addr = bldretry(retrybuf, 1);
    return;
}
```

SEE ALSO

freeexit, **bldretry**, **signal**

---

bldretry

Build System Retry Linkage via longjmp in SPE

SYNOPSIS

```
#include <bldexit.h>

void * bldretry(jmp_buf env, int code);
```

DESCRIPTION

bldretry is called to build linkage code enabling a program location defined with **setjmp** to be used as a retry routine. The **env** argument is a **jmp_buf**, which has been initialized by an earlier call to **setjmp**. (The **jmp_buf** type is defined in the header file **<setjmp.h>**. See "Program Control Functions" in Chapter 2 of the SAS/C Library Reference, Volume 1 for further information.) The **code** argument is an integer value to be returned by the resumed call to **setjmp**. If the value of **code** is 0, 1 is returned.

RETURN VALUE

bldretry returns the address of retry linkage code constructed by **bldretry**. This address should be supplied to the operating system as the address at which retry is to take place.

On completion of a successful retry, the effect is the same as the effect of a **longjmp(env,code)**.

CAUTIONS

The values in registers on entry to the retry routine are ignored. If you need to pass information from an exit routine to a retry routine, you should use other mechanisms, such as **extern** storage, for this purpose.

The retry linkage code is freed immediately before the **longjmp** to the location defined by **env** is performed. You must call **bldretry** again to perform another retry.

EXAMPLE

See the example for **bldexit**.

SEE ALSO

bldexit, **setjmp**

---

btrace

Generate Traceback in SPE

SYNOPSIS

```
void btrace(__remote void(*func)());
```

DESCRIPTION

Refer to Chapter 6, "Function Descriptions" in SAS/C Library Reference, Volume 1 for more details. Note that SPE does not support a 0 **func** address.

When **btrace** is called directly or indirectly from a **bldexit** exit function, the traceback will be incomplete unless the exit has stored the point of interrupt. See the description of **bldexit** for more information.

RETURN VALUE

btrace returns **void**.

PORTABILITY

btrace is not portable.

IMPLEMENTATION

The SPE implementation of **btrace** is in L$UBTRC.

---

dspserv

Dataspace services

SYNOPSIS

```
#include <osdspc.h>
int dspserv(char *request, ...);
```

DESCRIPTION

The **dspserv** function implements the functionality of the OS/390 assembler DSPSERV macro. The **request** argument is a null-terminated string. The request must be one of the following:

“CREATE” creates a new dataspace.

"RELEASE"	returns system resources used to contain user's data.
"DELETE"	deletes a dataspace.
"EXTEND"	increases the size of a dataspace.
"LOAD"	loads some area of a dataspace into central storage.
"OUT"	tells system that it can page some areas of a dataspace out of central storage.

The remainder of the argument list is a list of keywords followed, in most cases, by an argument specifying a value for the keyword. The list is terminated by the **_Dend** keyword. The supported keywords and their associated data are as follows:

- The **_Dreason** keyword is used to pass back a reason code in the event that the service fails. The next argument should be a pointer to an **int** that will contain the reason code as set in R0 by the DSPSERV macro if the DSPSERV macro returns non-0 in R15.
- The **_Dstoken** keyword is equivalent to the Assembler **STOKEN** keyword. The next argument should be the address of an 8-character array. The CREATE service returns the **STOKEN** in this array. All other services require the **STOKEN** as input.
- The **_Dname** keyword is equivalent to the Assembler **NAME** keyword. The next argument should be an 8-character array. This parm is required for the CREATE service. Depending on the setting of the **_Dgenname** parm, all or part of this name will be used to name the dataspace.
- The **_Dgenname** keyword is equivalent to the Assembler **GENNAME** keyword. The next argument should be one of the following strings:

"NO"	The name supplied in **_Dname** must be unique within the address space and will be used to name the dataspace.
"COND"	The name in **_Dname** will be used to name the dataspace unless it is already being used, in which case a unique name will be generated.
"YES"	The name in **_Dname** will be altered to force a unique name for the dataspace.

- The **_Doutname** keyword is equivalent to the Assembler **OUTNAME** keyword. The next argument should be an 8-character array. If specified, the name of the dataspace will be returned here if **_Dgenname** YES or COND was specified on a CREATE request.
- The **_Dstart** keyword is equivalent to the Assembler **START** keyword. The next argument should be a pointer that contains the beginning address of a block of storage in a dataspace. This parameter is required on RELEASE, LOAD, and OUT requests.
- The **_Dblkmax** keyword is related to the Assembler **BLOCKS** keyword. The next argument should be an integer value containing the maximum size in blocks that a new dataspace can grow to. The limit is 524,287 blocks. This parm should only be used with the CREATE service. This parm is optional, and if it is omitted, the default maximum size will be taken from site installation defaults or IBM defaults (at the time of this writing, 239 blocks.)
- The **_Dblkinit** keyword is related to the Assembler **BLOCKS** keyword. The next argument should be an integer value specifying the initial size (in blocks) of a dataspace. This parm should be used only with the CREATE service. This parm is optional, and if omitted, the default initial size will be taken from site installation defaults or IBM defaults (at the time of this writing, 239 blocks.)

- The **_Dblksize** keyword is related to the Assembler **BLOCKS** keyword. The next argument should be an integer value containing a size in blocks. This parm should used with one of the following services: RELEASE, EXTEND, LOAD, or OUT. This parm indicates the size of storage to be affected by one of these services.
- The **_Dttoken** keyword is equivalent to the Assembler **TTOKEN** keyword. The next argument should be the address of an 16-byte array that has been set to indicate a TCB. This is usually returned from a call to the TCBTOKEN macro. The CREATE service will use this field to assign the dataspace to a TCB and the DELETE service will use it to delete a dataspace owned by another TCB. This parameter is optional.
- The **_Dorigin** keyword is equivalent to the Assembler **ORIGIN** keyword. The next argument should be the address of a pointer that will be set to either 0 or 4096, indicating the lowest address of a new address space. This parm is optional for the CREATE service.
- The **_Dnumblks** keyword is equivalent to the Assembler **NUMBLKS** keyword. The next argument should be the address of an **int** where one of the following will be returned:

  for "CREATE" — the maximum size (in blocks) of the newly created dataspace.

  for "EXTEND" — the number of blocks by which the dataspace was extended.
- The **_Dranglist** keyword is equivalent to the Assembler **RANGLIST** keyword. The next argument should be the address of a *range list*. The range list consists of a number of entries (as specified by **_Dnumrange**) where each entry is 8 bytes long. The first four bytes of the entry contain the starting virtual storage address of the dataspace range to be released, and the second four bytes contain the number of pages in the dataspace range to be released. **_Dranglist** is an optional parameter of the RELEASE service.
- The **_Dnumrange** keyword is equivalent to the Assembler **NUMRANGE** keyword. The next argument should be an integer value representing the number of entries in the range list. **_Dnumrange** is an optional parameter of the RELEASE service.
- The **_Dvar** keyword is equivalent to the Assembler **VAR** keyword. The next argument should be one of the following strings:

  "YES" — If the system cannot extend the dataspace to the requested size, extend it to limits set by the system.

  "NO" — ABEND the caller if the size cannot be accommodated. **_Dvar** is an optional parameter of the EXTEND service. **_Dvar**, NO is the default.
- The **_Dend** keyword indicates the end of the list of keywords.

The following parms can be used by authorized programs only:

- The **_Ddref** keyword is equivalent to the Assembler **DREF** keyword. The next argument should be one of the following strings:

  "YES" — Disabled programs can access the dataspace.

  "NO" — Only enabled programs can access the dataspace. **_Ddref** is an optional parameter of the CREATE service.
- The **_Dscope** keyword is equivalent to the Assembler **SCOPE** keyword. The next argument should be one of the following strings:

  "SINGLE" — The dataspace may only be referenced by the owning address space.

"ALL" or "COMMON" — The dataspace may be referenced by many address spaces.

- The **_Dkey** keyword is equivalent to the Assembler **KEY** keyword. The next argument should be a pointer to a byte containing the storage key of the dataspace to be created. The key should be placed in bits 0-3 of the byte. **_Dkey** is an optional parameter of the CREATE service and if it is omitted, the key associated with the dataspace will be the same as that of the caller.
- The **_Dfprot** keyword is equivalent to the Assembler **FPROT** keyword. The next argument should be one of the following strings:

  "YES" — The dataspace will be fetch-protected; that is, access is limited to programs running key 0 or with the storage key with which the dataspace was created.

  "NO" — The dataspace will not be fetch-protected.
- The **_Ddisabled** keyword is equivalent to the Assembler **DISABLED** keyword. The next argument should be one of the following strings:

  "NO" — The caller is enabled for I/O and external interrupts. **_Ddisabled,** NO is the default.

  "YES" — Valid only for RELEASE requests of pages that reside in DREF storage.

RETURN VALUE

dspserv returns 0 if the DSPSERV macro was successful. If the DSPSERV macro fails, it returns the return code from the macro, which will be a positive value. **dspserv** may also return -1 to indicate an unknown or invalid keyword combination.

IMPLEMENTATION

The **dspserv** function is implemented by the source module L$UDSPC.

EXAMPLES

Example 1

This example creates a dataspace and initializes some data using mostly system defaults.

```
#include <osdspc.h>
#include <string.h>
char stoken[8];
void __far * pFar;
int rc;
int alet;
void *origin;

rc = dspserv("CREATE",
             _Dstoken,  stoken,
             _Dname,    "LSC",
             _Dorigin,  &origin,
             _Dend);
if (rc != 0)
   abend(1);
```

```
rc = aleserv("ADD",
             _Lstoken, stoken,
             _Lalet,   &alet,
             _Lend);
if (rc != 0)
   abend(2);


memcpy(&pfar, &alet, 4);
memcpy(((char *) &pfar)+4, &origin, 4);
strcpy(pFar, "DATASPACE STORAGE");
```

Example 2

This example extends a dataspace by 300 blocks.

```
#include <osdspc.h>
char stoken[8];
int rc;
int numblks;
int reason;


rc = dspserv("EXTEND",
             _Dstoken,  stoken,
             _Dreason,  &reason,
             _Dblksize, 300,
             _Dnumblks, &numblks,
             _Dend);
if (rc != 0)
   abend(1);
```

Example 3

This example deletes a dataspace.

```
#include <osdspc.h>
char stoken[8];
int rc;

rc = dspserv("DELETE",
             _Dstoken,  stoken,
             _Dend);
if (rc != 0)
   abend(1);
```

SEE ALSO

aleserv, **falloc**, **ffree**

---

exit

Terminate Execution in SPE

SYNOPSIS

```
#include <stdlib.h>

void exit(int code);
```

DESCRIPTION

exit terminates the program and returns control to its caller. The integer argument **code** is returned in register 15. The meaning or value of the exit code is subject to alteration by a start-up routine.

RETURN VALUE

Control does not return from **exit**.

PORTABILITY

exit is portable.

IMPLEMENTATION

The SPE implementation of **exit** is in L$UMAIN and L$UEXIT. If **blkjmp** is used, **exit** is implemented as a call to **longjmp**. However, **exit** does not call **longjmp** if the **longjmp** routine is not linked into the load module. In a multiple load module application, it may be desirable to modify L$UEXIT so that **longjmp** is always called.

SEE ALSO

atexit

---

falloc

Create a dataspace, or allocate storage from an existing dataspace

SYNOPSIS

```
#include <osdspc.h>
void __far * falloc(struct _DSPC ** dspc, size_t size);
```

DESCRIPTION

falloc allocates memory from a dataspace, optionally creating a new dataspace. **dspc** must be the address of a pointer. On first use, this pointer must be initialized to 0. On return, the pointer will contain the address of a _DSPC struct and a new dataspace will be created. The far pointer returned will address the first byte of this dataspace and the offset portion of this pointer will contain either 0 or 4096 depending on hardware. Subsequent calls to **falloc** that pass the address of a pointer addressing this _DSPC struct will allocate additional storage from the same dataspace. Each call will return an offset that is a multiple of the blocksize of the dataspace (4096). The length requested in size can be any value up to 2G, but should be a multiple of 4K to make the best use of the dataspace.

RETURN VALUE

falloc returns a far pointer that addresses the first byte of the new block of memory aligned on a 4K boundry. If **falloc** fails, the returned pointer will contain all zeros. Note that a valid far pointer may have zeros in the offset half of the far pointer.

IMPLEMENTATION

The **falloc** function is implemented by the source module L$UFALC.

EXAMPLE

```
#include <osdspc.h>
#include <string.h>
void __far * fptr;
pDSPC pdspc = 0;

fptr = falloc(&pdspc, 4096);

if (fptr)
   memcpy(fptr, "New dataspace");
```

SEE ALSO

dspserv, **aleserv**, **ffree**

---

ffree

Release dataspace resources

SYNOPSIS

```
#include <osdspc.h>
void ffree(void __far * fptr);
```

DESCRIPTION

ffree releases system resources associated with blocks of memory residing in a dataspace. The **fptr** far pointer must have been set by a previous call to **falloc**. If all allocations from a dataspace are released, then the DU-AL will be updated to remove the ALET for the dataspace, and the dataspace will be deleted.

RETURN VALUE

none

IMPLEMENTATION

The **ffree** function is implemented by the source module L$UFALC.

EXAMPLE

```
#include <osdspc.h>
#include <string.h>
void __far * fptr;
pDSPC pdspc = 0;

fptr = falloc(&pdspc, 4096);
```

```
memcpy(fptr, "New dataspace");

ffree(fptr);
```

SEE ALSO

dspserv, **aleserv**, **falloc**

---

format

Write Formatted Output to a String

SYNOPSIS

```
#include <lclib.h>

int format(char *s, const char * form , ...);
```

DESCRIPTION

format is similar to the **sprintf** function except that it does not support the floating-point conversions (**e**, **E**, **f**, **g**, **G**). Refer to Chapter 6, "Function Descriptions" in SAS/C Library Reference, Volume 1 for more information.

RETURN VALUE

format returns the number of characters written to the location addressed by **s**.

DIAGNOSTICS

If there is an error during output, **format** returns a negative number. The absolute value of this number equals the number of characters written up to the point of error or 1, if none are written.

PORTABILITY

format is not portable.

IMPLEMENTATION

format is implemented as a faster, smaller version of **sprintf**.

SEE ALSO

sprintf

---

free

Free a Block of Memory in SPE

SYNOPSIS

```
#include <stdlib.h>

void free(char *block);
```

DESCRIPTION

free frees a block of memory previously allocated by the **malloc** function. **block** is a pointer to the memory block.

ERRORS

See the IMPLEMENTATION section for the **malloc** function.

PORTABILITY

free is highly portable.

IMPLEMENTATION

Any **malloc** memory that has not been freed at program termination will be automatically freed.

See the IMPLEMENTATION section for the **malloc** function for more details about the implementation of **free**.

SEE ALSO

malloc, **DMSFRET**, **FREEMAIN**

---

freeexit

Free Exit Linkage Code in SPE

SYNOPSIS

```
#include <bldexit.h>

void freeexit(void *area);
```

DESCRIPTION

freeexit frees the memory associated with a **bldexit** exit routine. The argument to **freeexit** is the address returned by the previous call to **bldexit**.

RETURN VALUE

None.

CAUTION

Do not call **freeexit** for an exit routine that is defined to the operating system.

PORTABILITY

freeexit is not portable.

EXAMPLE

See the example for **bldexit**.

SEE ALSO

bldexit

---

getenv

Get Value of Environment Variable in SPE

SYNOPSIS

```
#include <stdlib.h>

char *getenv(const char *name);
```

DESCRIPTION

The **getenv** function searches an environment variable list for a variable **name** that matches the string pointed to by the **name** argument. See the description of the **putenv** function for a discussion on altering and creating environment variables.

RETURN VALUE

The **getenv** function returns a pointer to the string value associated with the matched name in the environment variable list. If a matching name is not found, **getenv** returns the value **NULL**.

CAUTIONS

A subsequent call to **getenv** overwrites the array pointed to by the first call.

Note that, in the SPE implementation, the names as well as the values of environment variables are case sensitive.

PORTABILITY

The **getenv** function is defined in accordance with the ANSI Standard for C and POSIX.1.

IMPLEMENTATION

The only environment variables accessible to the SPE **getenv** are those added by a previous call to **putenv** or **setenv**. Note that if you have a program called with **exec**-linkage that uses the **oemain** start-up routine, the environment variables passed by the caller of **exec** are added to the environment using **putenv** before control is passed to the initial function.

SEE ALSO

putenv, **setenv**

---

loadm

Dynamically Load a Load Module in SPE

SYNOPSIS

```
#include <dynam.h>

void loadm(char *name, __remote /* type */ **fpp());
```

DESCRIPTION

loadm loads the module named by the first argument string **name** and stores a C function pointer referencing the initial function of the module.

RETURN VALUE

loadm provides an indirect return value in the form of a function pointer that addresses the entry point of the loaded module. If the module is in the C language, calling the returned function always transfers control to the **_dynamn** function of the module.

If the module to be loaded cannot be found, a **NULL** pointer is stored in the location addressed by **fpp**.

ERRORS

Errors are implementation defined.

CAUTIONS

The second argument must be a pointer to an object declared as pointer to function returning (some C data type).

Subordinate load modules must be linked with the SPE version of the #DYNAMN (for reentrant load modules) or #DYNAMNR (for non-reentrant load module) code. This code is in L$UDYNM and L$UDYNR, respectively. The SPE implementation of **loadm** cannot load modules linked with the full library version of #DYNAMN or #DYNAMNR. Similarly, the full library implementation of **loadm** cannot load modules linked with the SPE version of #DYNAMN or #DYNAMNR.

PORTABILITY

loadm is nonportable.

IMPLEMENTATION

The SPE implementation of **loadm** is in L$ULDR. Under OS/390, **loadm** is implemented via SVC 8; under CMS, it is implemented via the NUCXLOAD command; and under CICS, it is implemented via the EXEC CICS LOAD command. Note that the SPE **loadm** is substantially less functional than that of the full library; therefore, this

version should be considered as a prototype only. Any serious use of **loadm** in SPE will require L$ULDR to be extended or rewritten.

---

malloc

Allocate Memory in SPE

SYNOPSIS

```
#include <stdlib.h>

char *malloc(size_t size);
```

DESCRIPTION

malloc allocates a block of dynamic memory of the size requested by **size**.

RETURN VALUE

malloc returns the address of the first character of the new block of memory. The allocated block is suitably aligned for storage of any type of data.

ERRORS AND DIAGNOSTICS

Errors are implementation defined. See IMPLEMENTATION below.

If adequate memory is not available or if 0 bytes were requested, a NULL (0) pointer is returned.

CAUTIONS

The contents of a block of memory on allocation are random.

The **realloc** function is not supported, and the full-library **realloc** function does not work with L$UHEAP. You may implement this function as a simple extension to the existing implementation.

PORTABILITY

malloc is highly portable.

IMPLEMENTATION

The SPE version of **malloc** is supplied in source as L$UHEAP. The following description of the function is based on this implementation.

The external variable **_heap** can be used to define the total amount of storage to be reserved for **malloc** allocation. By default, the amount reserved is determined by the start-up routine. Unlike the full-library **malloc**, this version does not attempt to allocate more storage if the initial amount is insufficient.

The memory management routines are simpler than those used in the full library and most suited to applications with simple memory management demands. If the application requires more complex memory management, modify L$UHEAP appropriately or consider using the operating system memory management directly with the DMSFREE or GETMAIN macros, or via EXEC CICS GETMAIN commands.

The **malloc** implementation in L$UHEAP respects two macros, CHECKING and SYNCH. If CHECKING is defined, code that checks for overlays of allocated memory is generated. If an overlay is detected, user ABEND 1206 is issued. If the **free** function detects an invalid argument, user ABEND 1208 is issued.

If SYNCH is defined, code is generated that allows **malloc** to be used in asynchronous exits. The supplied object code is compiled with SYNCH defined and CHECKING undefined.

Note that the **calloc** and the pool allocation functions are compatible with L$UHEAP. Refer to "Memory Allocation Functions" in Chapter 2 of SAS/C Library Reference, Volume 1 for more information.

SEE ALSO

free, **DMSFREE**, **GETMAIN**

---

oeabntrap

Trap ABENDs as USS Signals

SYNOPSIS

```
#include <oespe.h>

int oeabntrap(int code);
```

DESCRIPTION

oeabntrap is used to intercept OS/390 ABENDs and transform them into an appropriate USS signal. If the program is being debugged with **dbx** (or any other similar debugger), the debugger is informed of the ABEND and is allowed to recover it. **oeabntrap** is supported only in SPE programs; similar functionality is defined automatically when the standard C framework is used.

The **code** argument is a symbolic value indicating the particular function wanted, one of **TRAP_ON**, **TRAP_OFF**, or **TRAP_AUTO**. When **code** is **TRAP_ON**, the ABEND trapping functionality is enabled. When **code** is **TRAP_OFF**, ABEND trapping functionality is disabled. When **code** is **TRAP_AUTO**, ABEND trapping functionality is enabled, and an **atexit** routine is defined to disable ABEND trapping at the end of program execution.

If an ABEND occurs while ABEND trapping is enabled, the following events take place:

1. The **ptrace** system call is issued to inform any debugger of the event. If the debugger requests that the ABEND be recovered, an appropriate ESTAE retry is issued.
2. If the ABEND is not recoverable or was issued by the library, or if the signal from a previous ABEND is still pending, the ABEND is allowed to complete.
3. An ESTAE retry is issued.
4. The retry routine sends the ABENDing process a signal using the **kill** system call. If a debugger requested the signal, the signal is chosen by the debugger. If not, an appropriate signal is selected by the retry routine (SIGILL, SIGSEGV, or SIGFPE for program check ABENDs, SIGABRT for user ABENDs, SIGABND for system ABENDs).
5. If the signal is unable to be delivered, the process is terminated with a user ABEND 1225.

RETURN VALUE

oeabntrap returns **0** if its function was successful. It returns a positive value if the call had no effect (for instance, specifying **TRAP_OFF** before any trap was established). It returns a negative value if any other error occurred.

CAUTIONS

If you issue your own ESTAE macros in addition to using **oeabntrap**, it is your responsibility to make sure that your exits do not interfere with the operation of **oeabntrap**.

Note that you should not block any signal that might be generated by the ABEND trap. The effects of this are unpredictable and are likely to cause recursive ABENDs.

PORTABILITY

oeabntrap is not portable.

USAGE NOTES

Source for **oeabntrap** is supplied in SPE source modules L$UZABN and L$UZEST.

EXAMPLE

This example uses **oeabntrap** to catch ABENDs and defines a SIGSEGV handler to call the **btrace** function to show the location of the error.

```
#include <oespe.h>
#include <unistd.h>
#include <lclib.h>
#include <string.h>
#include <setjmp.h>
#include <signal.h>

jmp_buf ABEND_escape;
    /* where to run to after an ABEND */

static int ABEND_trapped;

void trace_out(char *line) {
    /* this function writes a btrace output line to file
       descriptor 2 */
    write(2, line, strlen(line));
    write(2, "\n", 1);
}

void ABEND_handler(int signum) {
    char buf[60];
    sprintf(buf, "Interrupted by signal %d!\n", signum);
    write(2, buf, strlen(buf));
    btrace(&trace_out);
    longjmp(ABEND_escape, 1);
}

int ptrvalid(int *ptr) {
    /* return whether storage addressed by ptr can be read */
```

```
    sigaction segv_action, prev_action;
    int ok;
    volatile int value;

    if (ABEND_trapped = 0) {
        oeabntrap(TRAP_AUTO);
          /* possibility of error ignored */
        ABEND_trapped = 1;
   }

    if (setjmp(ABEND_escape) != 0) goto failed;
          /* set up retry from handler */
    segv_action.sa_handler = &ABEND_handler;
    sigemptyset(segv_action.sa_mask);
    segv_action.sa_flags = 0;
    struct sigaction(SIGSEGV, &segv_action, &prev_action);
          /* we'll try to access the storage even if
             sigaction fails... */
    value = *ptr;    /* force reference to *ptr */
    ok = 1;          /* it must be valid */
    goto complete;
failed:
    ok = 0;          /* the pointer is no good */
complete:
    sigaction(SIGSEGV, &prev_action, 0);
                     /* restore previous SIGSEGV handling */
    return ok;
}
```

SEE ALSO

ESTAE, **sigaction**

---

putenv

Update Environment Variable in SPE

SYNOPSIS

```
#include <lclib.h>

int putenv(const char *string);
```

DESCRIPTION

The **putenv** function alters an environment variable's value or creates an environment variable with a name and value corresponding to the string pointed to by the **string** argument. The format of the **string** argument is

variable-name=value

variable-name
specifies the name of the variable to be created or updated.

=value
specifies a string assigned to the variable. It defaults to a null string `""` if it is not specified. All blanks are significant in the string.

RETURN VALUE

The **putenv** function returns **0** if successful.

PORTABILITY

The **putenv** function is not portable. It is a complementary extension to **getenv**.

SEE ALSO

getenv, **setenv**

---

setenv

Modify Environment Variables in SPE

SYNOPSIS

```
#include <stdlib.h>

int setenv(const char *name, const char *value)
```

DESCRIPTION

setenv adds or replaces environment variables. **name** is the name of the environment variable. **value** is the new value to be assigned to the environment variable.

RETURN VALUE

setenv returns **0** if it is successful. **setenv** returns **-1** if it is not successful.

CAUTION

If **name** includes an equal sign (=), **setenv** will fail.

PORTABILITY

setenv is defined in accordance with POSIX.1a.

SEE ALSO

getenv, **putenv**

---

unloadm

Dynamically Unload a Load Module in SPE

SYNOPSIS

```
#include <dynam.h>

void unloadm(__remote /* type */ (*fp)());
```

DESCRIPTION

unloadm unloads the executable module containing the function addressed by **fp**. If the module is no longer in use, **unloadm** deletes it from memory.

RETURN VALUE

unloadm does not have a return value.

ERRORS

Errors are implementation defined.

CAUTIONS

Attempting to call a function in an unloaded module is not recommended.

No provision is made for unloading modules at program termination automatically. However, this sort of functionality can be implemented in a function that is registered with **atexit**.

PORTABILITY

unloadm is nonportable.

IMPLEMENTATION

Refer to the IMPLEMENTATION section for **loadm**.

SEE ALSO

loadm, **atexit**

---

vformat

Write Formatted Output to a String

SYNOPSIS

```
#include <lclib.h>

int vformat(char *s, const char *form, va_list arg);
```

DESCRIPTION

vformat is equivalent to **format** with the variable argument list replaced by **arg**. The **arg** parameter has been initialized by a type **va_start** macro and possibly by

va_arg calls. **vformat** does not change the **va_arg** list pointers; for example, it does not use the **va_start**, **va_arg**, or **va_end** macros to process the variable argument list.

RETURN VALUE

vformat returns the number of characters written to the location addressed by **s**.

ERRORS AND DIAGNOSTICS

If there is an error during output, **vformat** returns a negative number. The absolute value of this number equals the number of characters written up to the point of the error.

PORTABILITY

vformat is not portable.

IMPLEMENTATION

vformat is implemented as a faster, smaller version of **vsprintf**.

EXAMPLE

```
#include <lclib.h>
#include <stdarg.h>

   /* Format an error message buffer via format  */
   /* Format the remaining buffer with vformat   */
void error (char *msg_buf, char *fname, *format, ...)
   va_list args;
   int msg_len;
   va_start(args, format);
   msg_len = format(msg_buf,"ERROR in %s: ", fname);
   if (msg_len > 0) msg_buf += msg_len;
   else msg_buf -= msg_len;
   vformat(msg_buf, format, args);
   va_end(args);
}
```

SEE ALSO

format, **sprintf**, **vsprintf**

Linking for SPE

Under OS/390

In general, the autocall (SYSLIB) input data set for SPE programs is a concatenation of these elements in the following order:

1 your own autocall libraries (including modified versions of SPE routines)

2 the data set SASC.SPEOBJ (the SPE object library)

3 the base resident library data set SASC.BASEOBJ.

You can use the SPE operand of the COOL CLIST or the ENV=SPE operand of the cataloged procedures to define these libraries in the correct order.

Under CMS

To create a MODULE file for a program using the SPE framework, issue the CMS GLOBAL command for these elements in the following order before issuing the LOAD and GENMOD commands:

1 your autocall TXTLIB(s) (including modified versions of SPE routines)

2 LC370SPE TXTLIB (the SPE TEXT library)

3 the base resident library LC370BAS TXTLIB.

You can use the SPE operand of the COOL EXEC to make these GLOBAL libraries in the correct order.

Under CICS

The autocall (SYSLIB) input data set for CICS SPE programs is a concatenation of these elements in the following order:

1 your own autocall libraries (including modified versions of CICS SPE routines)

2 the data set SASC.CICS.SPEOBJ (the CICS SPE object library)

3 the base resident library data set SASC.BASEOBJ.

You can use the LCCCL cataloged procedure under OS/390 and specify the symbolic parameters ENV=CICS.SPE and ENTRY=CSPE to define these libraries in the correct order and to select the correct entry point. You can also use the CICS and SPE operands in the COOL clist on TSO to perform the same function.

If you are developing your CICS SPE application under CMS, you can specify the CICS and SPE operands when you invoke the COOL EXEC. The resulting object code must be shipped to the OS system containing the CICS system on which you plan to run. The object code must also be link-edited with the CICS command-level stubs.

Use the following linkage editor control statements when you build your load module:

```
INCLUDE SYSLIB (DFHEAI)
INCLUDE SYSLIB (DFHEAI0)
ORDER DFHEAI
```

where the DDname SYSLIB points to the CICS load library containing the command-level stubs.

If your SPE application is targeted for a CICS/VSE system and you are using INDEP, you must include manually the VSE version of L$UPREP that is named L$UPREPD.

Caution

If your program calls a function that is not supported in the SPE framework and you use the standard resident library data set as an autocall library, no error occurs when the program is linked. At execution time, the library may issue a user ABEND 1212 or there may be other, unpredictable results. Refer to "The SPE Library" earlier in this chapter for information on the functions that can be used in the SPE framework.

CHAPTER 15

Developing Applications for Use with UNIX System Services OS/390

Introduction

UNIX System Services (USS) OS/390 provides all the traditional services of the OS/390 operating system plus many new services. Two prominent features of these new services are the USS hierarchical file system (HFS) and the MVS/ESA USS Shell and Utilities. The SAS/C Compiler enables you to develop applications that can be invoked from either the traditional OS/390 environment or the USS shell.

- Applications that reside in an OS/390 data set are invoked with the CALL command from the TSO/E environment or with JCL statements and the SUBMIT command in a batch environment. They can also be invoked from the shell using the **`pdscall`** shell command.
- Applications that reside in an USS HFS file are usually invoked directly from the USS shell, and can be run either interactively or in the background.

The USS hierarchical file system and shell provide an operating system interface that complies with the POSIX 1003.1 standard. With Release 6.00, the SAS/C Library provides a number of functions that give you the capability of exploiting the functionality of this interface. USS OS/390 and the SAS/C Library also implement portions of the draft POSIX 1003.1a standard. The support provided for these two standards is referred to as POSIX.1 and POSIX.1a in this documentation.

The POSIX.1 standard defines an operating system interface and environment that is based on the UNIX operating system. Many of the commands and features of the USS shell will be familiar to you if you have studied UNIX. The portable operating system interface (POSIX) was designed to support application portability at the source level.

What is a POSIX Application?

A POSIX application can be considered to be any application that takes advantage of the POSIX.1 and POSIX.1a support provided by the SAS/C Library. As will be explained later in this chapter, there are various levels of conformance to the POSIX.1 standards, ranging from strict conformance to a rather loose conformance that takes advantage of the extensions provided by the SAS/C Library. A POSIX application may have any of these levels of conformance to the POSIX.1 standards.

POSIX References

The SAS/C documentation does not describe the POSIX.1 standard or cover the general background concepts required to successfully develop a POSIX application. For information on these topics, please refer to the following publications:

ISO/IEC 9945-1: 1990 (IEEE Std 1003.1-1990)
Information technology–Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API) [C Language]

Zlotnick, Fred (1991) *The POSIX.1 Standard: A Programmer's guide*, The Benjamin/Cummings Publishing Company, Inc.

POSIX Conformance

The POSIX.1 standard specifies three levels of conformance:

Strictly Conforming POSIX.1 Applications
A strictly conforming C language POSIX.1 application uses only those facilities described in ISO/IEC 9945 and the ISO C language Standard.

Conforming POSIX.1 Applications
There are two categories for this type of conformance:

- An ISO/IEC conforming C language POSIX application uses only the facilities described in ISO/IEC 9945 and the approved C language bindings for any ISO or IEC standard.
- A <National Body> conforming C language POSIX application uses only the facilities described in ISO/IEC 9945 and the approved C language bindings for any ISO or IEC standard or the specific standards of a single ISO/IEC member, such as BSI (British Standards Institute).

Conforming POSIX.1 Applications Using Extensions
This type of application uses documented, nonstandard language extensions that are consistent with the POSIX.1 standard. An example of this type of application would be a program that uses the POSIX.1 interface but also uses the SAS/C Socket Library for TCP/IP.

While SAS/C supports the compilation and execution of strictly conforming POSIX.1 programs, it is also intended to support mixed-mode programming. That is, SAS/C

Release 6.00 supports the production of applications that combine POSIX functionality, such as pipes and directories, with traditional OS/390 and SAS/C functionality like VSAM. SAS/C is also flexible about how POSIX and non-POSIX features can be combined. UNIX oriented programs can be written with small nonportable sections to exploit OS/390 features, and OS/390 oriented programs can be written which exploit POSIX functionality when appropriate.

Strictly Conforming POSIX Programs

A POSIX program is strictly conforming if it uses only ISO/ANSI and POSIX standard library features and does not depend on any undefined or implementation-defined behavior. If you want a program to strictly conform to the POSIX.1 standards, you should define the feature test macro **_POSIX_SOURCE** before including any standard header file. (One way to define this symbol is using the **define** compiler option.) When **_POSIX_SOURCE** is defined, the only declarations included in standard header files are those of standard symbols. This ensures that any symbols defined by your application will not conflict with any non-POSIX extensions defined in the standard header files.

The symbol **_POSIX1_SOURCE** can be defined to the value 2 to define a program that is strictly conforming except for its use of features from the POSIX.1a draft standard. If **_POSIX1_SOURCE** is defined to 2, then **_POSIX_SOURCE** need not be defined.

POSIX Programs with Extensions

Many programs use POSIX functionality but are not intended to be strictly conforming. For instance, a socket application cannot be strictly conforming, since sockets are not defined by the POSIX.1 or POSIX.1a standard. If the feature test macro **_POSIX_SOURCE** is defined, then critical declarations (for instance, the type **fd_set**) will be omitted from headers like **<sys/types.h>**, which will cause a socket application to fail to compile.

SAS/C Release 6.00 defines the feature test macro **_SASC_POSIX_SOURCE** to specify POSIX functionality plus SAS/C functionality. If the symbol **_SASC_POSIX_SOURCE** is defined before inclusion of the first system header file, then ISO/ANSI and POSIX header files will define ISO/ANSI and POSIX functionality, plus SAS/C extensions. If you define one of **_POSIX_SOURCE** or **_POSIX1_SOURCE** as well as **_SASC_POSIX_SOURCE**, the standard feature test macro has precedence, that is, SAS/C extensions will not be defined.

If you do not define any feature test macro, then POSIX header files may include definitions of SAS/C extensions. However, ISO/ANSI header files will define only ISO/ANSI sanctioned symbols. Therefore, if you want to use POSIX functionality defined in ISO/ANSI header files (for example, the **fdopen** routine declared in **<stdio.h>**), you should define a feature test macro.

Note that the preferred method of defining the **_SASC_POSIX_SOURCE** feature test macro is to specify the **posix** option at compile time.

For more information about feature test macros and the SAS/C Library header files, refer to Chapter 1, "Introduction to the SAS/C Library," in the SAS/C Library Reference, Volume 1.

Portability Considerations

The POSIX.1 standard is designed to facilitate the portability of programs in source form; however, compliance with the standard does not guarantee programs will be completely portable. Dealing with ASCII to EBCDIC character translation is just one of

the many issues you should be familiar with if you are developing POSIX applications that will be ported to a platform other that the IBM System/370. Refer to the discussion of data interchange formats in The POSIX.1 Standard: A Programmer's Guide for a detailed treatment of this subject.

Compiling POSIX Programs

The **posix** compiler option modifies compiler behavior in order to establish certain defaults required by the POSIX.1 standard. A program compiled without the **posix** option may not behave completely according to the standard even if the program's code is completely conforming.

Some of the effects of the **posix** option are:

- The feature test macro **_SASC_POSIX_SOURCE** is automatically defined.
- The **refdef** option is assumed.
- The special POSIX variable names **environ** and **tzname** are made automatically **__rent** unless explicitly declared **__norent**.
- A run-time flag is set defining the object module as POSIX-compiled.

When a load module containing a **main** function is link-edited, the resulting load module is considered to be POSIX compiled if any constituent object module was compiled with the **posix** option. (For this reason, if you are writing routines that may be used in both POSIX and non-POSIX programs, you should not compile them with the **posix** option, because this would force any load modules that use them to be considered POSIX.)

If the main load module of a program is defined as POSIX compiled, certain library defaults are changed in order to bring them into conformance with the POSIX standard. For instance, in a program that is not POSIX compiled, the function call **fopen("sysin", "r")** opens the file associated with the DDname SYSIN. In a POSIX compiled program, this call opens the USS HFS file **"sysin"** in the current directory.

exec-Linkage Programs

An executable load module may be stored in either a partitioned data set (PDS) or an USS hierarchical file system (HFS) file. When a load module is stored in a PDS, it can be loaded and invoked by standard OS/390 supervisor calls (SVCs), but there is no POSIX defined way to invoke it. When a load module is stored in the HFS, it is inaccessible to OS/390 SVCs but can be executed by means of the USS **exec** system call. **exec** is used by the USS shell to call its commands, as well as by other POSIX applications that need to pass control to other programs. A program that is given control by **exec** rather than by an OS/390 SVC is called an **exec**-linkage program.

Note: A program will have **exec**-linkage if it is stored in the USS HFS. You can either direct the output from the linkage editor directly to the HFS or move the load module from a PDS to the HFS using the USS OPUT, OGET, or OCOPY commands. Also, the SAS/C **pdscall** utility can be used to invoke a program stored in a PDS with **exec**-linkage. △

exec-linkage is not required for a program to use POSIX functionality. For instance, a program that reads the HFS can be run in TSO. However, because TSO is not a POSIX conforming environment, certain POSIX behavior details are not implemented in TSO. For instance, the POSIX standards require that when a program is given

control, **stdin**, **stdout**, and **stderr** are defined to be POSIX file descriptors 0, 1, and 2. TSO does not set up these standard file descriptors, and **stdin**, **stdout**, and **stderr** reference the TSO terminal instead, in this environment. If this particular behavior is important to your application, the application load module should be stored in the HFS and invoked with **exec**, in order to guarantee the proper behavior.

Whether or not a program has **exec**-linkage affects a number of details of run-time library behavior. Specific instances are discussed later in this chapter.

Using the USS Shell

You can compile, link, and run SAS/C applications directly from the USS shell as was described in the following chapters:

- Chapter 5, “Compiling C Programs,” on page 81
- Chapter 7, “Linking C Programs,” on page 131
- Chapter 8, “Executing C Programs,” on page 171

In addition to this basic information, you should also be familiar with shell scripts and the **make** utility if you are developing applications under the USS shell.

Shell Scripts

USS shell scripts provide an efficient means of executing a frequently used series of commands. For example, you may have a series of commands that you enter frequently to either compile or back up your SAS/C applications–a shell script can be a very efficient method of automating this process. See the IBM MVS/ESA OpenEdition MVS User’s Guide (SC23-3013-01) for information about writing USS shell scripts.

make Utility

The USS make utility is used to manage the software development process. It enables you to define a *makefile* that specifies the relationship between the various source and object files used in your application. The makefile is then used by the **make** utility to remake the application as necessary to update the object files after a source file is changed. See MVS/ESA OpenEdition MVS Advanced Application Programming Tools (SC23-3017-01) from IBM for more information about the **make** utility.

File Access

SAS/C POSIX applications can access either HFS files or OS/390 data sets from the USS shell. This section describes how this is accomplished. For detailed information about file access and input/output considerations, refer to Chapter 3, "I/O Functions," in the SAS/C Library Reference, Volume 1. Also refer to the MVS/ESA OpenEdition MVS User’s Guide (SC23-3013-01) for general information about file access and USS.

Hierarchical File System (HFS) Files

The USS Hierarchical File System (HFS) is patterned after the UNIX file system. All files are located in directories, and the directories are organized in a hierarchical

manner with each directory being a subdirectory to another directory until you reach the root directory.

When you start a shell session, a process is created. Each process maintains a location in the HFS. This location is called the *working directory*. The initial working directory you are placed in when you start a shell session is called your *home directory*. You can use the USS shell command **cd** to change the current working directory location for the shell.

Pathnames are used to specify the location of files within the directory structure. A pathname starts with the root directory and works its way down the directory hierarchy, separating each directory name with a single slash (/), until you come to the name of the file. For example, the following pathname specifies a file named **qsort.c** located in the **src** subdirectory of the **userxyz** directory.

```
/u/userxyz/src/qsort.c
```

Notice that the **userxyz** directory is a subdirectory of the **u** directory, which is located in the root directory. The root directory is signified by the single slash (/) at the beginning of the pathname. This type of pathname, which shows the complete path from the root directory to the file, is called an *absolute pathname*.

There is a second type of pathname, called a *relative pathname*, that specifies a path relative to your current working directory. To specify a relative pathname for a file, simply enter the pathname to the file from your current location in the HFS. For example, if the current working directory is **userxyz**, the **qsort.c** file could specified as follows:

```
src/qsort.c
```

Notice that the beginning slash (/) is not used in a relative pathname.

The following special fields can also be used when specifying a relative pathname:

. is used to specify the current directory.

.. is used to specify the parent directory.

For example, we could have specified the **qsort.c** file from the **userxyz** directory in either of the following ways:

```
./src/qsort.c
../userxyz/src/qsort.c
```

To put all this in the context of a SAS/C program, we could open the **scrambled.txt** file from our **qsort.c** program with any of the following statements, provided the current working directory is **/u/userxyz/src**.

```
datafile = fopen("./scrambled.txt", O_RDWR);

datafile = fopen("../scrambled.txt", O_RDWR);

datafile = fopen("../src/scrambled.txt", O_RDWR);
```

We could also use the following absolute pathname to specify the **scrambled.txt** file:

```
datafile = open("./scrambled.txt", O_RDWR);
```

Note: The exact format of the filename specification depends upon whether or not the **posix** option was used at compile time. △

In this case the **scramble.txt** file will be found no matter where the working directory is when the program is executed.

OS/390 Data Sets

OS/390 data sets can also be accessed from an **exec**-linkage program running under the USS shell. If a **//** precedes a filename, the filename is assumed to be either of the **tso** styles. For example, the following statement could be used to reference a PDS member from the shell:

```
datafile = fopen("//scramble.text(eggnog)", "r+");
```

In an **exec**-linkage program compiled with the **posix** option, you must precede the filename with **//** even if you are using a style prefix such as **tso:**. If the filename does not begin with exactly two slashes, it will be interpreted as an HFS file. For example, the following statement will attempt to open the file named **tso:scramble.text(eggnog)** in the current working directory of the HFS:

```
datafile = fopen("tso:scramble.text(eggnog)", "r+");
```

Obviously, this is not the desired result. To correctly open the **userid.scramble.text(eggnog)** PDS member, you must precede the **tso:** with two slashes.

Note: This convention of using two slashes to access OS/390 data sets or CMS files from your **exec**-linkage programs cannot be used with USS shell commands. For example, you cannot use **//** to concatenate a PDS member with an HFS file using the **cat** command. △

Accessing the Transient Library

In an ordinary (batch/TSO) OS/390 environment, SAS/C Library routines needed at runtime are loaded from the transient library. This library is located in one of three ways: it may be allocated to a STEPLIB (or tasklib) data set, it may be allocated to the DDname CTRANS, or it may reside in linklist/LPALIB.

When an application is called by the shell (or, more generally, invoked by the POSIX **exec** system call), it runs in an address space that has no preallocated DD statements. This creates problems for transient library access under the shell. The SAS/C Library solves this problem as follows:

1. If the environment variable ddn_CTRANS is defined when a SAS/C program begins execution, the value of the variable is assumed to be an OS/390 data set name, which is dynamically allocated to the DDname CTRANS.
2. When a SAS/C program running with an allocated CTRANS calls the **fork** function, the same file is allocated to CTRANS in the child before **fork** returns.
3. When a SAS/C program running with an allocated CTRANS performs an **exec**, the environment variable ddn_CTRANS is generated to contain the name of the CTRANS data set, unless this variable already exists. This variable is passed to the called program, so that if it is a SAS/C compiled program, it will have access to the same CTRANS data set.

When a program is linked with the all-resident library, it normally does not access CTRANS; however, if such a program issues an **exec** call, the CTRANS data set name is still recorded in the ddn_CTRANS environment variable, since the called program might require transient library access.

It is recommended that you update **/etc/profile** so that ddn_CTRANS is set to an appropriate value automatically whenever the shell starts up.

USS supports a feature similar to the SAS/C CTRANS support for STEPLIB data sets, using the environment variable STEPLIB. If the value of STEPLIB is CURRENT,

an existing STEPLIB data set is propagated on **exec**. Alternately, STEPLIB may name one or more data sets to be allocated to STEPLIB in the new address space. This support will also work for access to the SAS/C Transient Library.

Processes

In the USS shell environment, a running program is called a process. Multiple processes can be executed independently of each other, with each process receiving its own address space. To facilitate the referencing of these independent processes, each process is associated with a process identification number (PID).

A process can be used to start other processes. The USS shell is a process itself that can be used to start multiple processes. At any one time, the shell can have one process running in the *foreground* and several processes running in the *background*. A foreground process ties up the shell and prevents you from entering additional commands while it is running. The shell does not wait for the completion of background processes; they run in the background in a manner similar to an OS/390 batch job. The **ps** shell command can be used to display the PID for the shell and all of the processes running under the shell.

Processes can also be started by SAS/C programs. The **exec** family of functions and the **fork** and **atfork** functions are commonly used to start a new process. The SAS/C extension function **oeattach** can be used to start a new process in the same address space as the old process, which may offer improved performance over the use of **fork** and **exec**. Refer to the SAS/C Library Reference, Volume 2 for more information about these functions.

User and Group Identification

The USS shell environment assigns a user identification number (UID) to each user. A user can also belong to one or more groups of users, each of which is assigned a group identification number (GID). These identification numbers are used to assign file ownership and control access privileges. Read, write, and execute permission is assigned to each file by either the file owner or the system administrator. Access privileges can be assigned to the user, the user's groups, and others.

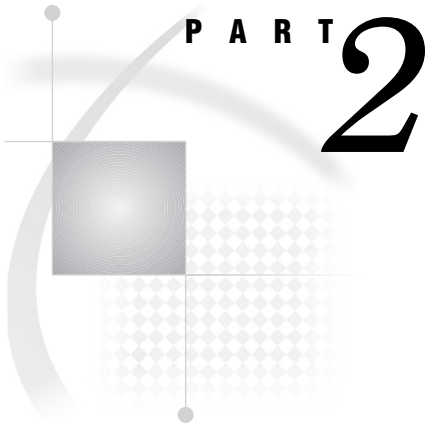
Every process has a real and an effective UID, as well as a real and an effective GID. When you start a shell session, the real and effective UIDs are set to your user identification number, and the real and effective GIDs are set to your group identification number.

The effective UIDs and GIDs are used to control file access, and the real UIDs and GIDs are used for accounting purposes. File access is determined as follows:

- If the effective UID of a process matches the UID of the file's owner, then the process will have user access privileges.
- If the effective GID of a process matches the GID of the file's owner, then the process will have group access privileges.
- If neither the effective UID or GID of the process match those of the file's owner, then the process will have others access privileges.

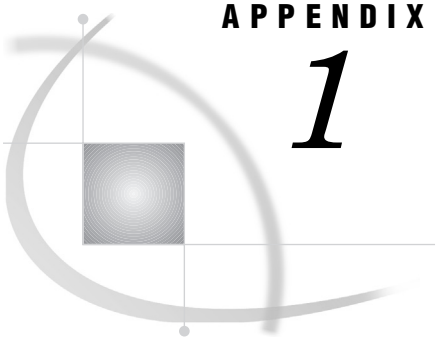
If a process executes another process, the access rights are normally determined by the effective UID and GID of the calling process and not the access privileges of the owner of the executable. This can create undesirable situations, such as the situation in which a running program must have the ability to change a file that you do not want the user who executed the program to be able to modify directly. This problem is

overcome by allowing a program to be defined to execute with the UID or GID of the program owner rather than that of the calling process. Also, suitably authorized programs can use the **`setuid`** and **`setgid`** functions to change the current process's effective UID and GID respectively.



PART 2

Appendixes



APPENDIX

1

The DSECT2C Utility

Introduction

This appendix describes the DSECT2C utility program. DSECT2C converts an assembler language dummy section, known as a DSECT, to an equivalent C structure definition. DSECT2C can be very helpful when writing C programs that interface with assembler language programs. The first section of this appendix explains how to use DSECT2C, including the required operating system commands or control language. The second section discusses the C **`typedefs`** and macro definitions that are generated along with the structure definition.

How to Use DSECT2C

This section explains how to use the DSECT2C utility. Included are discussions of input and output files, options, usage notes, and the operating system commands or control language required to invoke DSECT2C.

Input and Output

The input file to DSECT2C is an assembler listing file (under OS/390, this is the SYSPRINT data set; under CMS, it is the filetype LISTING file). To generate this file, create an assembler language program consisting of only the DSECT definition and an

END instruction. Example Code 16.1 on page 344 shows a sample assembler language input file. If the DSECT is in an existing assembler language source library, it can be included with a COPY instruction, followed by an END instruction. DSECTs in macro libraries can be allowed to expand in the listing.

Example Code A1.1 Sample Assembler Language Input File

```
SAMPLECB DSECT
NAME     DS    CL8
ADDRESS  DS    A
NUMBER   DS    F
         ORG   ADDRESS
DNUM     DS    D
         ORG ,
FLAGS    DS    XL1
ADDR3    DS    AL3
         END
```

Invoke the assembler and produce a listing file. DSECT2C expects **`stdin`** to be redirected to the listing file.

DSECT2C writes the C structure definition to **`stdout`**. This file also includes any **`typedefs`** used in the structure, a set of macro definitions for the structure members, and other output as specified by DSECT2C options.

Example Code 16.2 on page 344 shows the output file produced by DSECT2C for the DSECT defined in Example Code 16.1 on page 344. There are four distinct parts:

- C **`typedefs`** corresponding to assembler language types
- a C structure corresponding to assembler language DSECT
- C macros for each of the fields in the structure
- optional cross-reference information.

Example Code A1.2 Sample DSECT2C Output File

```
#if !defined(_AL3)
#define _AL3
typedef struct
   {
   char BF : 24;
   } AL3;
#endif

#if !defined(_CL8)
#define _CL8
typedef char CL8(|8|);
#endif

struct SAMPLECB
   {
   CL8 name;
   union
      {
      struct
         {
```

```
         void *address;
         int number;
         } _s0;
      double dnum;
      };
   char flags;
   AL3 addr3;
   };

#define ADDRESS                         _s0.address
#define ADDR3                           addr3
#define DNUM                            dnum
#define FLAGS                           flags
#define NAME                            name
#define NUMBER                          _s0.number

/*
SYMBOL      OFFSET    LENGTH    TYPE    C-TYPE    C-NAME
ADDRESS     000008    000004    A       void *    _s0.address
ADDR3       000011    000003    AL3     AL3       addr3
DNUM        000008    000008    D       double    dnum
FLAGS       000010    000001    XL1     char      flags
NAME        000000    000008    CL8     CL8       name
NUMBER      00000C    000004    F       int       _s0.number
SAMPLECB    000000    000000
*/
```

Note: DSECT2C generates identifiers for unnamed fields and inner structures when necessary. These identifiers always have the format **_f** or **_s** followed by an integer number. △

Options

DSECT2C accepts six options, as shown in Table A1.1 on page 346.

Table A1.1 DSECT2C Options

Option	Explanation
-c	Assembler language comments in the input file are included as C comments in the output file.
-d	DSECT2C generates a declaration of the structure, instead of a definition. The DSECT name in lowercase is used as the identifier. For example, without the **-d** option, the format of the structure generated by DSECT2C is **struct CNTLBLOK {** **.** **.** **.** **};** If the **-d** option is used, DSECT2C changes that to **struct CNTLBLOK {** **.** **.** **.** **} cntlblok;**
-i	The assembler language instruction associated with each structure member is included as a comment in the output file. The **-i** option automatically enables the **-c** option.
-n	DSECT2C assigns names to any unions within the output structure. These names are of the form **_un** where *n* is an interger
-u	**stderr** output is in uppercase.
-x	DSECT2C adds cross-reference information to the output file.
-z	DSECT2C subtitles **LENGTH_ZERO** or **LENGTH_ZERO_REF** macros for arrays with 0 elements and **LENGTH_ZERO_2D** macros for two-dimensional arrays whose first dimensions are 0.

Usage Notes

1 DSECT2C creates structures that use anonymous unions, nonaligned structures, and noninteger bitfields. These types are nonstandard extensions to the C language and may not be supported by other compilers. C source files that include these structure definitions may need to be compiled with the **bitfield** option, using **char** as the default allocation unit.

For more information about language extensions, see “Language Extensions” on page 28. For more information on compiler options, see Chapter 6, “Compiler Options,” on page 101.

2 DSECT2C changes certain national characters to a digraph that is acceptable in a C identifier. The characters and their associated digraphs are as follows:

```
@       A_
#       P_
$       D_
```

Note that only leading $’s are changed. Use the **dollars** compiler option if a structure member identifier contains embedded $’s.

3 DSECT2C may or may not be able to detect erroneous input. Do not use DSECT2C on a DSECT that caused the assembler to produce WARNING or ERROR messages.

4 DSECT2C calculates the offset and alignment required for each DSECT field. Each field in the DSECT is assumed to be properly aligned for its type. (Improperly aligned fields cause the assembler to generate a WARNING message.)

5 DSECT2C ignores all instructions in the input file except for those composing the DSECT itself and EQU statements that precede the DSECT.

6 Any instructions in the DSECT that are prevented from appearing in the assembler listing file (by the PRINT instruction, for example) will not appear in the C structure DSECT2C creates.

7 If the assembler used to create the input listing file has been modified to create a special listing format, DSECT2C may not be able to find the DSECT instruction or instructions in the DSECT. In this case, use an editor to remove unnecessary records in the listing before invoking DSECT2C.

8 DSECT2C converts assembler language constructs with a duplication factor of 0 to an array with 0 elements. For example, consider this assembler language statement:

```
BEGINBUF   DS   0C
```

By default, DSECT2C converts this to the following structure member:

```
char beginbuf(|0|);
```

The SAS/C Compiler accepts such an array declaration within a structure definition as a language extension. However, most C compilers do not allow a declaration of an array with 0 elements. If you attempt to compile a program containing a structure generated by DSECT2C that contains this kind of declaration using another compiler, that compiler will probably generate an error message and refuse to compile the program. The **-z** option causes DSECT2C to generate **LENGTH_ZERO**, **LENGTH_ZERO_2D**, and **LENGTH_ZERO_REF** macros, which can be used to overcome this problem.

The **-z** option causes DSECT2C to substitute a call to the **LENGTH_ZERO** macro instead of generating an array of 0 elements. The **LENGTH_ZERO** macro accepts two arguments, the type and name of the member. For example, if the **-z** option is used, DSECT2C converts the assembler language statement shown above (**BEGINBUF DS 0C**) to the following statement:

```
LENGTH_ZERO(char, beginbuf);
```

You must supply a definition of the **LENGTH_ZERO** macro that generates a declaration for the member that is acceptable to the target compiler.

LENGTH_ZERO_2D macro

The **LENGTH_ZERO_2D** macro is used when the member is a two-dimensional array whose first dimension is 0. As shown here, the **LENGTH_ZERO_2D** macro accepts three arguments, the type of the member, its name, and the second dimension:

```
LENGTH_ZERO_2D(int, nfield, 2);
```

LENGTH_ZERO_REF macro

The **LENGTH_ZERO_REF** macro is generated by DSECT2C in the C macro corresponding to a structure member. By default, the macro for a zero-length array member is defined as follows:

```
#define BEGINBUF      beginbuf(|0|)
```

If the **-z** option is used, DSECT2C substitutes the following **LENGTH_ZERO_REF** macro:

```
#define BEGINBUF      LENGTH_ZERO_REF(beginbuf)
```

The **LENGTH_ZERO_REF** macro accepts one argument, the member name.

DSECT2C also generates default definitions of these three macros. These definitions produce the same member declaration as would be generated if the **-z** option were not in effect. Each definition is protected by an **#if !defined** preprocessor statement, so any definitions you supply override the default definitions. The default definitions generated by DSECT2C are

```
#if !defined(LENGTH_ZERO)
#define LENGTH_ZERO(type, name) type name(|0|)
#endif
#if !defined(LENGTH_ZERO_2D)
#define LENGTH_ZERO_2D(type, name, dim) type name(|0|)(|dim|)
#endif
#if !defined(LENGTH_ZERO_REF)
#define LENGTH_ZERO_REF(name) name(|0|)
#endif
```

Invoking DSECT2C

DSECT2C is a C program and can be invoked as you would any other C program. In general, you must specify the name of the DSECT to be converted and any options, redirect **stdin** to the assembler listing file, and redirect **stdout** to the desired output file.

In TSO

In TSO, you invoke DSECT2C with the CALL command, with the C command, or as a TSO command. If you use the TSO CALL command, invoke DSECT2C as follows:

```
CALL 'library.name(DSECT2C)'  'dsect [listing] output options'
```

library.name is the name of the data set containing DSECT2C.

dsect is the name of the DSECT to be converted.

listing is either the DDname allocated to the assembler listing or the data set name of the assembler listing. If a data set name is specified, it must be prefixed with either **tso:** or **dsn:**. The highest level qualifier of the data set must also be included.

output is either the DDname allocated to the output data set or the data set name of the output data set.

options are DSECT2C options.

Following is an example:

```
CALL 'SASC.LOAD(DSECT2C)'
        'IHADCB <tso:MYDSECT.LIST(IHADCB) >tso:MYDSECT.H(IHADCB)'
```

Consult your SAS Software Representative for C compiler products for the correct library.name.

Under CMS

Under CMS, invoke DSECT2C as follows:

```
DSECT2C dsect [listing ]output options
```

dsect	is the name of the DSECT to be converted.
listing	is the assembler listing file.
output	is the output file.
options	are DSECT2C options.

Following is an example:

```
DSECT2C FSCBD <FSCBD.LISTING >FSCBD.H
```

Under OS/390 batch

The DSECT2C cataloged procedure can be used to execute DSECT2C under OS/390 batch. This procedure allocates SYSPRINT to a temporary data set, invokes the assembler, and then invokes DSECT2C to produce the C structure. You should provide a DD card for the assembler SYSIN data set and a DD card for DSECT2C's D2COUT data set. Example Code 16.3 on page 349 shows typical JCL for using the DSECT2C cataloged procedure.

Example Code A1.3 Sample JCL for Running the DSECT2C Cataloged Procedure

```
//SAMPD2C  JOB job card information
//*-------------------------------------------------
//*  INVOKE ASSEMBLER AND DSECT2C
//*
//*  REPLACE GENERIC DATA SET NAMES AS APPROPRIATE
//*-------------------------------------------------
//STEP1        EXEC DSECT2C,PARM.D2C='dsect-name options '
//ASM.SYSIN    DD DSN=assembler.source.file ,DISP=SHR
//D2C.D2COUT   DD DSN=your.output.file ,DISP=OLD
//
```

In this example,

dsect-name	is the name of the DSECT to be converted.
options	are DSECT2C options. Separate options with blanks, not commas.

The ASM step of the DSECT2C procedure contains a SYSLIB DD card for the source file libraries SYS1.MACLIB and SYS1.AMODGEN. Note that DSECT2C is not invoked unless the return code from the assembler is 0.

The DSECT2C procedure contains the JCL shown in Example Code 16.4 on page 349.

Example Code A1.4 Expanded JCL for DSECT2C

```
//DSECT2C  PROC
//ASM      EXEC PGM=ASMBLR,PARM=TERM
//SYSTERM  DD SYSOUT=*
//SYSPUNCH DD DUMMY
//SYSLIN   DD DUMMY
```

```
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(TRK,(5,1))
//SYSUT2   DD DSN=&&SYSUT2,UNIT=SYSDA,SPACE=(TRK,(5,1))
//SYSUT3   DD DSN=&&SYSUT3,UNIT=SYSDA,SPACE=(TRK,(5,1))
//SYSLIB   DD DSN=SYS1.MACLIB,DISP=(SHR,KEEP,KEEP)
//         DD DSN=SYS1.AMODGEN,DISP=(SHR,KEEP,KEEP)
//SYSPRINT DD DSN=&&LISTING,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1089),
//            DISP=(NEW,PASS),UNIT=SYSDA
//D2C      EXEC PGM=DSECT2C,COND=(0,NE,ASM)
//STEPLIB  DD DSN=SASC.LOAD,DISP=SHR        COMPILER LIBRARY
//         DD DSN=SASC.LINKLIB,DISP=SHR       RUNTIME LIBRARY
//SYSIN    DD DSN=*.ASM.SYSPRINT,DISP=(OLD,DELETE,DELETE),
//            VOL=REF=*.ASM.SYSPRINT
//SYSTERM  DD SYSOUT=*
//SYSPRINT DD DDNAME=D2COUT
```

typedefs and Macros

This section discusses the C **typedefs** and macro definitions that are generated along with the structure definition.

Converting Assembler Language Types to C Language Types

Assembler language supports a much larger variety of types than does the C language. In addition, the best representation of a specific assembler language type often depends on how the data are accessed or modified. Therefore, conversion of assembler language types to C types is not always straightforward. For this reason, DSECT2C constructs **typedefs** for some types. This allows a clearer definition of some types, as well as making it easier for the programmer to choose a different C type if necessary. **CL8** and **AL3**, shown in Example Code 16.2 on page 344, are examples of such **typedefs**.

Since the **typedef** name can only be declared once, DSECT2C encloses each **typedef** in preprocessor statements to prevent multiple declarations of the **typedef** name from occurring when two or more structures are included in the same source file.

Using Symbol Macros

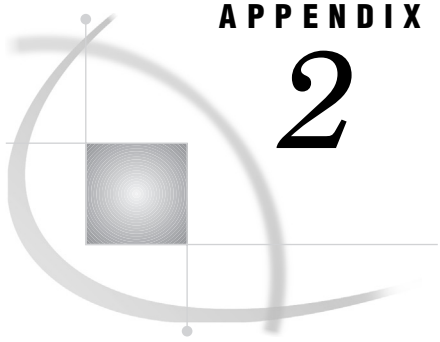
As part of the conversion, DSECT2C generates a macro for each symbol in the DSECT. The replacement list for each macro is the member identifier associated with the symbol. Using these macros to refer to the member is usually easier (and more readable) than coding the member identifier itself. Example Code 16.2 on page 344 shows examples of such macros.

For example, given the structure **SAMPLECB** defined in Example Code 16.2 on page 344, you can code the following:

```
#include "samplecb.h"
int f(struct SAMPLECB *s)
   {
   return s->NUMBER; /* better than 's->_s0.number' */
   }
```

Messages

The SAS/C Compiler and Library Quick Reference Guide contains a list of all DSECT2C diagnostic messages. All error conditions cause termination of execution.



APPENDIX

2

The AR370 Archive Utility

Introduction

The AR370 archive utility is a program that is used to create or maintain a collection of object modules called an *AR370 archive*. AR370 archives have a file organization that permits them to be used as COOL autocall libraries, supporting autocall of extended names.

Logically speaking, an AR370 archive is organized as a collection of *members*, identified by a member name that resembles a filename. The member names serve only to identify the members to the AR370 utility. Otherwise, member names are not significant and do not affect the autocall process. For each object module contained in an AR370 archive, the AR370 utility records the names of external symbols defined or referenced in the member (including external objects with extended names). This allows COOL to find the member that defines a particular symbol. No connection is needed between an AR370 member name and the external symbol names defined by the member.

Physically, each AR370 archive is composed of three parts:

header
: contains information such as the date of the last modification and the release number of the AR370 utility that made the modification.

member archive
: contains a copy of each file added to the library. (For AR370 archives, unlike OS/390 partitioned data sets, the order of members may be significant.)

symbol table
: contains a list of each external symbol defined or referred to by any member of the archive.

When adding or replacing members, the AR370 utility inserts a copy of each input file into the member archive. The utility also searches the external symbol dictionary (ESD) of each input file, creates a sorted list of ESD entries, and inserts the list in the library symbol table. The library symbol table is used by COOL to search an archive efficiently for ESD symbols and extended names.

Using the AR370 Archive Utility under CMS

Under CMS, the AR370 archive utility is invoked directly with the following command:

```
AR370 cmds [posname] arcname [fname...]
```

The *cmds* argument must be specified and consists of an optional hyphen (**-**), followed by one of the command characters **d**, **m**, **r**, **x**, or **t**. (t may be specified in combination with any other command.) Optionally, you can concatenate the command character with one or more of the command modifier characters **a**, **b**, **e**, **f**, **j**,**q**, or **v**. The command and command modifier characters are described later in this section.

The optional *posname* argument specifies the name of a specific archive member and is required only if one of the relative positioning command modifiers is specified.

The *arcname* argument specifies the archive fileid and must be present. The default filetype is **A**. If filetype is specified, it must be joined to the filename with a period (.). The default filemode is *. If a filemode is specified it must also be joined to the filetype with a period (.).

Each *fname* argument specifies the fileid of a file to be added or replaced or the name of an archive member to be manipulated. When used to specify fileids, the filename and filetype must be joined with a period (.). The default filemode is *. If a filemode is specified, it must also be joined to the filetype with a period (.). Member names must be specified exactly as they appear in the archive. When a file is added or replaced in the archive, the input archive member name is the *filename.filetype* translated to upper case.

One common use of the AR370 archive utility is to replace or add files to an archive. In the following example, the utility is invoked to replace the members RUN.TEXT and WALK.TEXT in the AR370 archive named ZOOM A, and verbose output is requested.

```
AR370 RV ZOOM RUN.TEXT WALK.TEXT
```

If either RUN.TEXT or WALK.TEXT does not exist in the ZOOM archive, they are added to the archive by the **r** command character.

When performing an add or replace, the CMS wildcard character, an asterisk (*), may be used in the filename or filetype for CMS style pattern matching; that is, all files matching the pattern are added or replaced. If you need a member name other than the filename, you can use the following syntax:

```
filename.filetype.filemode=membername
```

All of the *filename*, *filetype*, and *filemode* must be specified, and wildcards cannot be used. The new membername may be any name, but it is strongly recommended that it be a valid CMS filename, as members whose names are not valid CMS filenames are

difficult to manipulate with the AR370 utility. Here is an example of the use of replace-as:

```
AR370 R MYLIB THIS.TEXT.A=THAT.OBJECT
```

This command stores the file THIS.TEXT.A in the archive MYLIB with member name THAT.OBJECT.

Command Characters

The following command characters are recognized:

d deletes the specified members from the archive.

m moves the specified members. By default, the members are moved to the end of the archive. If an optional positioning character (**a** or **b**) is used, the *posname* argument must be present to specify that the named members are to be placed after (**a**) or before (**b**) *posname*. Note that the members are moved in the order of their appearance in the archive, not in the order specified on the command line. This means that when a number of members are moved, they remain in the same order relative to each other as before the move.

r replaces the specified files in the archive, creating new members for any that are not already present. If an optional positioning character (**a** or **b**) is used, the *posname* argument must be present to specify that the new members are to be placed after (**a**) or before (**b**) *posname*. In the absence of a positioning character, new members are appended at the end. When the **r** command character is used, the AR370 archive utility creates an archive file if it does not already exist. If no files are specified by *fname* arguments, the utility creates an empty archive.

t types a description of the contents of the archive. If no member names are specified, all members in the archive are described by name. If any member names are specified, information appears about those members only. Additional information is produced when either the (**v**) or (**e**) command modifiers is specified.

x extracts the named archive members. If no names are specified, all members of the archive are extracted. The member name is used as the name of each extract output file. The extract command does not alter or delete entries from the library.

Optional Modifier Characters

The following optional modifier characters are recognized:

a After: Positions the members to be moved or replaced after the member specified by the *posname* argument. If you specify **a**, you must specify *posname*.

b Before: Positions the members to be moved or replaced before the member specified by the *posname* argument. If you specify **b**, you must specify *posname*.

e Enumerate: Lists the defined symbols for the members specified for the type command. This modifier is meaningful only when used with

the type (**t**) command. When used with the verbose (**v**) command modifier, all defined and referenced symbols in the specified members are displayed.

f Files: On OS/390, specifies the use of a DDname prefix by the AR370 utility. Refer to the **files** option in Chapter 6, "Compiler Options," on page 101 for more information.

j Japan or uppercase: Produces all terminal output in uppercase (**japan**).

q Quick: Processes members of existing archives more quickly. This option keeps AR370 from reprocessing every member in the archive. It greatly reduces the amount of I/O needed to add, replace, delete, and move members in an archive, since no work file is used. You should use this option with care, however, because an existing library containing data could be destroyed if space in the data set runs out. Prior to using AR370 with the **q** option, it is recommended that you back up the archive so that you will not lose your data in the event that the original data set is destroyed.

The **q** option causes the member order to be maintained only in the symbol table. This avoids the I/O needed to reposition the actual objects within the archive. Only the order of members in the symbol table is relevant to the linker. Therefore, the order of the actual object files in the archive does not always have to be maintained. If an archive has been modified by AR370 and is subsequently changed without the **q** option, the actual order of the objects within the archive is changed to match the order of the members in the symbol table.

v Verbose: When used with the type (**t**) command, the **v** command modifier produces a long listing of information for each specified member in the form of name, date, size, and number of symbols. If no members are specified, a listing is produced for all members in the archive.

When used with the **d**, **m**, or **x** operations, the **v** modifier causes the AR370 archive utility to print each command operation character and the member name associated with that operation. For the **r** operation, the AR370 archive utility shows an **a** if it adds a new file or an **r** if it replaces an existing member. The verbose modifier also produces the AR370 archive utility's title and copyright notice.

y Yes: List the mangled name along with the demangled name. The **y** modifier is meaningful only when used with the **e** (**enumerate**) optional modifier.

Combinations of command and command modifier characters

Only the combinations of commands and command modifiers shown in the following table are meaningful.

Table A2.1 Command and Command Modifier Combinations

Command	Accepted Modifiers and Commands
d	**e, f, j, q, t, v**
m	**e, f, j, q, t, v** and **a | b**

Command	Accepted Modifiers and Commands
r	**e, f, j, q, t, v** and **a | b**
t	**d, e, f, j, m, r, v, x**
x	**e, f, j, t, v**

Using the AR370 Archive Utility in TSO

This section describes the LC370 and AR370 CLISTs, which allow you to use the AR370 Archive Utility in TSO. It also describes the OS/390 file attributes and DCB characteristics of AR370 archives.

The AR370 CLIST

The AR370 CLIST invokes the AR370 archive utility in TSO. The syntax is as follows:

```
AR370 archive action target [optional-parms]
```

The *archive* argument specifies the name of the AR370 archive. The *action* and *optional-parms* arguments are listed in the following sections.

action arguments

The *action* argument can be any of the following:

DELETE
deletes the AR370 archive member specified by the *target* argument.

DISPLAY
displays information about the AR370 archive member specified by the *target* argument.

EXTRACT
extracts the AR370 archive member specified by the target argument into the data set specified with the parameter **INTO**. For example, the following command extracts member GREEN from *userid*.SPECTRUM.A into the OS/390 file *userid*.COLOR.OBJ(GREEN).

```
AR370 SPECTRUM EXTRACT GREEN INTO(COLOR.OBJ(GREEN))
```

INCLUDE
uses the file specified by the *target* argument as a list of INCLUDE statements.

MOVE
moves the AR370 archive member specified by the *target* argument.

QUICK
processes members of existing archives more quickly. This option keeps AR370 from reprocessing every member in the archive. It greatly reduces the amount of I/O needed to add, replace, delete, and move members in an archive, since no work file is used. You should use this option with care, however, because an existing library containing data could be destroyed if space in the data set runs out. Prior to using AR370 with the **QUICK** option, it is recommended that you back up the archive so that you will not lose your data in the event that the original data set is destroyed.

REPLACE
adds or replaces the file specified by the *target* argument in the AR370 archive.

The *target* argument specifies the AR370 archive member or OS/390 data set to perform the action on. If the *action* is **REPLACE** or **INCLUDE**, the *target* argument is the name of a data set containing the replacement member or INCLUDE statements. When the AR370 utility adds or replaces a member on OS/390, the archive member name is determined by the input filename. If the input file is a PDS member, the archive member name is the same as the PDS member name. If the input file is a sequential data set, the member name is formed from the final two qualifiers of the input data set name. If the *action* is **MOVE**, **EXTRACT**, **DELETE**, or **DISPLAY**, the target name is an AR370 archive member name. For the **DISPLAY** action, you can use an asterisk (*) to specify all AR370 archive members.

optional-parms arguments

The *optional-parms* argument can be any of the following:

AFTER(*member-name*)
specifies the AR370 archive member after which the moved or replaced member is to be stored.

BEFORE(*member-name*)
specifies the AR370 archive member before which the moved or replaced member is to be stored.

DBCLIST
allows debugging of the AR370 CLIST.

INTO(*dataset-name*)
specifies the name of a TSO data set in which to store the extracted member. If the data set belongs to another user, the fully qualified name of the data set must be specified, and the name must be preceded and followed by three single quotes as follows:

```
INTO(''FRED.EXTRACT.OBJ''')
```

OTHER(*parameters*)
allows miscellaneous parameters to be passed to the AR370 archive utility.

PRINT(*dataset-name*)
specifies the name of a TSO data set in which to store the printed output from the AR370 archive utility. If the data set belongs to another user, the fully qualified name of the data set must be specified, and the name must be preceded and followed by three single quotes, for example, PRINT('"PROJECT.AR.LIST"').

SYMBOLS
requests display of the symbols that are defined in the members listed by the **DISPLAY** action. When used with the **VERBOSE** parameter, a listing of all symbols (both defined and referenced) is produced.

UPPER
specifies that the output from the AR370 archive utility is to be uppercase.

VERBOSE
specifies verbose information. Refer to the verbose (**v**) command modifier under "Optional Modifier Characters" on page 355 for more details.

WORKSPC('*primary secondary*')
specifies the workfile space allocation. The primary argument specifies the primary allocation, and the secondary argument specifies the secondary allocation.

WORKUNIT(*storage-unit*)
specifies the workfile allocation unit: **track**, **cylinder**, or **block**.

The LC370 CLIST

The LC370 CLIST has been enhanced to allow the compiler's object code output to be stored in an AR370 archive. The name of the archive in which the object code is to be stored is specified using the **arlib** compiler option. If the archive belongs to another user, its name must be given in full, and the name must be preceded and followed with three single quotes. For example:

```
'''userid.archive-name.AR'''
```

If the archive name is not quoted and does not have a final qualifier of AR, a final qualifier of AR is appended.

The **member** compiler option is used with the LC370 CLIST to specify the output archive member name, which must also be a valid OS/390 partitioned data set member name. If **member** is omitted and the input file is a partitioned data set member, the same member name will be used in the AR370 archive. If **member** is omitted and the input file is not a partitioned data set member, you will be required to enter a member name. Note that you may specify only one of the OBJECT and ARLIB keywords.

Here is an example of the use of the **arlib** and **member** compiler options:

```
LC370 RECIPE(BROIL) ARLIB(KITCHEN) MEMBER(BROILING)
```

In this example, the object code resulting from the compilation of **userid.RECIPE.C(BROIL)** is stored as member **BROILING** of **userid.KITCHEN.AR**.

OS/390 File Attributes

AR370 creates archives with RECFM FBS file attributes. In earlier releases, AR370 created archives with RECFM U file attributes. If you process an archive with RECFM U file attributes, it obtains an F record format. This record format allows a single, nonconcatenated archive to be processed by COOL without creating virtual copies of the archive members, thus improving prelinker performance.

To concatenate old archives with RECFM U with new archives having an F or FBS record format, specify the DCB characteristics of the concatenation to be RECFM=U.

If the concatenation is not RECFM U and it contains files with RECFM U and RECFM F and/or RECFM FBS, the following error message may be issued:

```
IEC024I INCONSISTENT RECORD FORMATS F AND U, ddn-n,dsname
```

You can also convert old style archives to new style archives by performing an add, move, or delete operation on the old archive with AR370.

DCB Characteristics

AR370 archives on OS/390 have the following DCB characteristics:

```
"RECFM=FBS,RECLEN=4080,BLOCKSIZE=4080"
```

Using the AR370 Archive Utility under OS/390 Batch

This section describes how to use the AR370 archive utility in batch mode under OS/390.

The AR370 Cataloged Procedure

The AR370 cataloged procedure is used to run the AR370 utility under OS/390 batch. The job shown in Example Code 17.1 on page 360 adds four object files from two OS/390 data sets to an AR370 archive and displays the archive information on the new members on SYSPRINT.

Example Code A2.1 Sample AR370 Archive Utility Batch Job

```
//RUNAR370 JOB job card information...
//GO EXEC AR370,PARM.AR='RTV'
//AR.SYSARLIB DD   DISP=OLD,DSN=userid.PROJ.AR
//AR.SYSIN    DD   *
 INCLUDE  MYOBJ(mem1,mem2,mem3)
 INCLUDE HEROBJ
//MYOBJ       DD   DISP=SHR,DSN=userid.PROJ.OBJ
//HEROBJ      DD   DISP=SHR,DSN=group.LEADER.OBJ(mem)
```

The JCL for the AR370 procedure shown in Example Code 17.2 on page 360 is correct as of the publication of this guide. However, it may be subject to change.

Example Code A2.2 JCL for AR370 Cataloged Procedure

```
//AR370     PROC
//*************************************************************
//*  NAME:  AR370                                    (AR370)  ***
//*  PRODUCT: SAS/C                                           ***
//*  PROCEDURE: OBJECT CODE ARCHIVAL                          ***
//*  DOCUMENTATION: TECHNICAL REPORT C-112                    ***
//*  FROM:  SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC    ***
//*************************************************************
//*
//AR       EXEC PGM=AR370#
//STEPLIB  DD DISP=SHR, COMPILER LIBRARY
//           DSN=SASC.LOAD
//         DD DISP=SHR, RUNTIME LIBRARY
//           DSN=SASC.LINKLIB

//SYSPRINT DD SYSOUT=A
//SYSTERM DD SYSOUT=A
//SYSARWRK DD UNIT=SYSDA,SPACE=(CYL,(2,1))
```

AR370 JCL requirements

If you choose to write your own JCL to run AR370 under OS/390, some or all of the following data definitions are required.

SYSARLIB	AR370 archive
SYSIN	INCLUDE cards used to specify new archive members
SYSPRINT	standard output
SYSARWRK	alternate data set for work space
SYSPUNCH	where extracted archive members are to be written
SYSTERM	standard error output

Additional DD statements are required if referenced by SYSIN INCLUDE statements.

AR370 PARM String

Under OS/390 batch, the syntax for the AR370 PARM string is as follows:

```
cmds [files_prefix] [posname] [memname ...]
```

cmds must be present. It is specified as a string of command characters and modifier characters. These specifications are the same as for the AR370 utility under CMS, as described previously in "Command Characters" on page 355 and "Optional Modifier Characters" on page 355 with the following exception:

CAUTION:

Specify the Member Name Under OS/390 batch, if you specify **x** without also specifying the member names, the members are extracted to the same flat file. Each extracted member is overwritten by the following extracted member, and at completion, the flat file will contain only the last extracted member. △

Additionally, OS/390 supports a command modifier character of **f**, which specifies the use of a DDname prefix by the AR370 utility.

files_prefix is required only if the **f** command modifier character is specified. The prefix can contain from one to three characters. The prefix then replaces the string SYS in all AR370 DDnames (except for SYSTERM, which is defined by the C library as the standard error file).

posname specifies the name of a specific archive member and is required only if one of the relative positioning command modifiers (**a** or **b**) is specified.

memname specifies the members of the archive to be processed. Member names must be specified exactly as they appear in the archive.

AR370 INCLUDE Statements

When the **r** command character is specified, AR370 reads SYSIN for INCLUDE statements that specify the locations of new or replacement archive members. The format of the INCLUDE statement is the following:

```
INCLUDE ddname [(member [,member])]
```

where the *ddname* may be allocated to a sequential data set, a PDS, or a member of a PDS. For example:

INCLUDE COBJ
: may refer to a sequential data set or one specific member of a PDS.

INCLUDE COBJ(MEM)
: refers to member MEM of a PDS.

INCLUDE COBJ(MEM1,MEM2)
: refers to members MEM1 and MEM2 of a PDS.

In OS/390 when a member is added to (or replaced in) an AR370 archive, the archive member name is determined from the name of the input file. If the input file is a PDS member, the archive member name is the same as the PDS member name. Otherwise, the archive member name is formed from the last two qualifiers of the input data set name.

If you need an archive member name different from the name that the AR370 utility would select by default, you can use the "replace as" feature of AR370. This feature lets you attach an archive member name to a DDname or member specification using the equal sign (=). For example:

```
INCLUDE COBJ:MEM2.OBJ
INCLUDE COBJ(MEM1=#BAZ,QUUX=*WOBBLE*)
```

The first of these statements stores the contents of the COBJ DDname as member MEM2.OBJ of the archive. The second of these two statements stores member MEM1 of the COBJ PDS as archive member #BAZ and PDS member QUUX as archive member *WOBBLE*.

The LC370CA Cataloged Procedure

The OS/390 JCL procedure LC370CA runs the compiler and stores the resulting object module in an AR370 archive. This procedure is useful if you are using extended names and want to have references to the compiled module resolved automatically by COOL. See Example Code 17.3 on page 362 for typical JCL to run LC370CA.

Example Code A2.3 Sample JCL for Compiling with Procedure LC370CA

```
//COMPILE JOB job card information
//*-------------------------------
//*     COMPILE A C PROGRAM AND STORE OBJECT IN AR-LIBRARY
//*     REPLACE GENERIC NAMES AS APPROPRIATE
//*-------------------------------
//STEP1     EXEC  LC370CA,PARM.C='options',
//          MEMBER=ar-member
//C.SYSIN    DD  DISP=SHR,DSN=your.source.library(member)
//C.libddn   DD  DISP=SHR,DSN=your.macro.library
//A.SYSARLIB DD  DISP=OLD,DSN=your.ar.archive
//
```

When you use LC370CA, you only need to provide DD cards for SYSIN (your source data set) and SYSARLIB (your output AR370 archive). You must also specify the MEMBER= option to specify the member name under which the object code should be stored in the output archive. This name must be a valid OS/390 PDS member name.

The LC370CA procedure contains the JCL shown in Example Code 17.4 on page 362. This JCL is correct as of the publication of this report. However, it may be subject to change.

Example Code A2.4 Expanded JCL for LC370CA

```
//LC370CA PROC MEMBER=DO.NOT.OMIT
//*************************************************************
//*  PRODUCT: SAS/C                                          ***
//*  PROCEDURE:  COMPILATION                                 ***
//*  DOCUMENTATION: SAS/C COMPILER AND LIBRARY USER'S GUIDE   ***
//*  FROM: SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC     ***
```

```
//*************************************************************
//*
//C       EXEC PGM=LC370B
//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(&MEMBER),SPACE=(3200,(10,10,1)),
//             DISP=(NEW,PASS),
//             UNIT=SYSDA,DCB=(RECFM=FB,LRECL=80,DSORG=PO)
//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
//A         EXEC PGM=AR370#,PARM=R,
//             COND=(4,LT,C)
//STEPLIB   DD DSN=SASC.LOAD,
//             DISP=SHR COMPILER LIBRARY
//          DD DSN=SASC.LINKLIB,
//             DISP=SHR RUNTIME LIBRARY
//SYSTERM   DD SYSOUT=*
//SYSPRINT  DD SYSOUT=*
//SYSARLIB  DD DSN=&&AR,SPACE=(4080,(10,10)),
//             DISP=(NEW,PASS),UNIT=SYSDA
//OBJECT    DD DSN=*.C.SYSLIN,VOL=REF=*.C.SYSLIN,DISP=(OLD,PASS)
//SYSIN     DD DSN=SASC.BASEOBJ(AR@OBJ),
//             DISP=SHR
```

The LCCCPCA Cataloged Procedure

The procedure LCCCPCA preprocesses and compiles a C program for CICS and stores the resulting object module in an AR370 archive. This procedure is useful if you are using extended names and want to have references to the compiled module resolved automatically by COOL. See Example Code 17.5 on page 363 for typical JCL to run LCCCPCA.

Example Code A2.5 Sample JCL for Compiling with Procedure LCCCPCA

```
//COMPILE JOB job card information
//*--------------------------------
//*     PREPROCESS AND COMPILE A C PROGRAM FOR CICS
//*     AND STORE OBJECT IN AR-LIBRARY
//*     REPLACE GENERIC NAMES AS APPROPRIATE
//*--------------------------------
//STEP1       EXEC LCCCPCA,PARM.C='options',
//            MEMBER=ar-member
```

```
//CCP.SYSIN   DD   DISP=SHR,DSN=your.source.library(member)
//C.libddn    DD   DISP=SHR,DSN=your.macro.library
//A.SYSARLIB  DD   DISP=OLD,DSN=your.ar.archive
//
```

When you use LCCCPCA, you only need to provide DD cards for SYSIN (your source data set) and SYSARLIB (your output AR370 archive). You must also use the MEMBER= option to specify the member name under which the object code should be stored in the output archive. This name must be a valid OS/390 PDS member name.

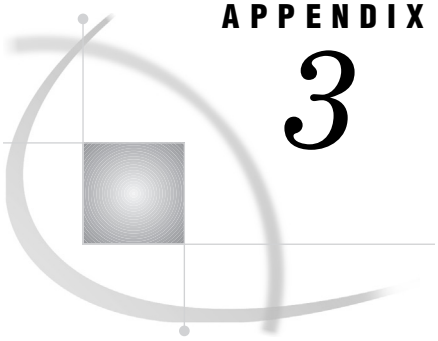
The LCCCPCA procedure contains the JCL shown in Example Code 17.6 on page 364. This JCL is correct as of the publication of this report. However, it may be subject to change.

Example Code A2.6 Expanded JCL for LCCCPCA

```
//LCCCPCA PROC MEMBER=DO.NOT.OMIT
//*************************************************************
//*  NAME: LCCCPCA                                 (LCCCPCA)    ***
//*  SUPPORT:  C COMPILER DIVISION                              ***
//*  PRODUCT: SAS/C                                             ***
//*  PROCEDURE:  CICS TRANSLATION/COMPILATION                   ***
//*  DOCUMENTATION: SAS/C CICS USER'S GUIDE                     ***
//*  FROM: SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC       ***
//*************************************************************
//*
//CCP      EXEC PGM=LCCCP0,REGION=1536K
//STEPLIB  DD DSN=SASC.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(&MEMBER),SPACE=(3200,(10,10,1)),
//            DISP=(NEW,PASS),
//            UNIT=SYSDA,DCB=(RECFM=FB,LRECL=80,DSORG=PO)
//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
//A         EXEC PGM=AR370#,PARM=R,
//            COND=((4,LT,CCP),(4,LT,C))
//STEPLIB   DD DSN=SASC.LOAD,
//            DISP=SHR COMPILER LIBRARY
//          DD DSN=SASC.LINKLIB,
//            DISP=SHR RUNTIME LIBRARY
//SYSTERM   DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSARLIB DD DSN=&&AR,SPACE=(4080,(10,10)),
//            DISP=(NEW,PASS),UNIT=SYSDA
//OBJECT    DD DSN=*.C.SYSLIN,VOL=REF=*.C.SYSLIN,DISP=(OLD,PASS)
//SYSIN     DD DSN=SASC.BASEOBJ(AR@OBJ),
//            DISP=SHR
```



APPENDIX

3

The AR2UPDTE and UPDTE2AR Utilities

Introduction

The utilities AR2UPDTE and UPDTE2AR transform an AR370 archive into a file that is suitable for input to the IBM IEBUPDTE utility and vice versa. These utilities can be useful for converting an existing object code PDS into an AR370 archive and for creating an OS/390 PDS from an existing archive.

AR2UPDTE Utility

AR2UPDTE is a utility program that converts an AR370 archive to an IEBUPDTE input format data file. AR2UPDTE reads in the archive and creates a new file of IEBUPDTE input format data. The AR2UPDTE output file can be used as input to the IBM IEBUPDTE utility to build an OS/390 partitioned data set that approximates the AR370 archive provided as input to AR2UPDTE. Together AR2UPDTE and IEBUPDTE can be used to copy every member of an AR370 format archive into a corresponding member of a partitioned data set.

Archives built on a non-OS/390 system may have member names that are not acceptable as member names to IEBUPDTE. AR2UPDTE offers a translation feature that permits the user to specify how archive member names should be translated to PDS member names. Default translation rules are always applied unless the user specifies that no translation should be performed.

Using AR2UPDTE under CMS

Under CMS, the AR2UPDTE utility is invoked directly with the following command:

```
ar2updte [options] infile outfile
```

options specifies one or more options, each of which is a single character preceded by a hyphen (-). Some options (for example, **-t**) must be followed by an option argument. The argument may be separated from the option by white space, but this is not a requirement. Note that the case of option characters is not significant, but that case is significant for most option arguments.

The following options are recognized:

-t `c:s`
: specifies a translation rule to be used by AR2UPDTE when deriving a PDS member name from an archive member name. More than one **-t** option can be specified. The option argument `c:s` indicates that if the string `'c'` (which can be longer than a single character) occurs in an archive member name, it is to be replaced by the string `'s'` in the output PDS member name.

  Unless **-x** is specified, default member translation rules are used. See the section "Default Member Translation Rules" on page 370 for details.

-x
: specifies that no character translations will be applied to the member names during the archive to IEBUPDTE conversion. The **-x** argument is optional. The **-x** option can be used to preserve the original input archive's member names, even if they do not conform to the IEBUPDTE rules for acceptable PDS member names. The resulting output may not be usable as input to IEBUPDTE, but it can be used as input to UPDTE2AR to build a copy of the input archive.

The *infile* and *outfile* arguments must be specified. The *infile* argument specifies the archive file identifier. It must be a valid archive. The *outfile* argument specifies the file identifier of the resulting output file, which is in IEBUPDTE input format.

Note: Under CMS, the default filetype for the *infile* argument is A. If filetype is specified, it must be joined to the filename with a period (.). The default filemode for the *infile* argument is *. The *outfile* argument specifies the file identifier of the resulting output file which is in IEBUPDTE input format. The default filetype for the *outfile* argument is IEBUPDTE. If filetype is specified, it must be joined to the filename with a period (.). The default filemode for the *outfile* argument is *. △

The following examples show typical AR2UPDTE command lines under CMS:

ar2updte testlib.a test.iebupdte
: Create a new IEBUPDTE input format file named **test.iebupdte** from the archive **testlib.a**.

ar2updte -x testlib.a test2.iebupdte
: Create a new IEBUPDTE input format file named **test2.iebupdte** from the archive **testlib.a** without performing any translations on the names of object members in the archive.

ar2updte -t ?:QU -t x:$ testlib.a test3.iebupdte
: Create a new IEBUPDTE input format file named **test3.iebupdte** from the archive **testlib.a**. Convert all question marks to the letters QU, and convert all x's to the dollar sign.

Using AR2UPDTE in TSO

This section describes calling AR2UPDTE in TSO using the TSO CALL command. Use the following syntax:

```
CALL 'library.name(AR2UPDTE)' '[options] infile outfile' ASIS
```

Here, *library.name* is the name of the library containing AR2UPDTE. Consult your SAS/C Software Representative for C compiler products for the correct library name.

See the section "Using AR2UPDTE under CMS" on page 368 for a description of the *options*. Note that *infile* and *outfile* will be interpreted as DDnames unless a SAS/C style prefix is used.

The following example shows a typical use of AR2UPDTE in TSO:

```
CALL 'SASC.LOAD(AR2UPDTE)' 'tso:testlib.a tso:testpds.data' ASIS
```

Using AR2UPDTE under OS/390 Batch

This section describes how to run AR2UPDTE under OS/390 batch using the AR2UPDTE cataloged procedure.

The AR2UPDTE cataloged procedure is used to execute AR2UPDTE under OS/390 batch. You must provide a SYSARLIB DD statement defining the input AR370 archive and a SYSPUNCH DD statement defining the output file. The output file must be defined to have fixed-record format with 80-byte records. If you need to pass special AR2UPDTE options, specify the OPTS keyword in the EXEC statement, as shown below. See the section "Using AR2UPDTE under CMS" on page 368 for information on the AR2UPDTE options and their meanings.

Example Code 18.1 on page 369 shows typical JCL for using the AR2UPDTE cataloged procedure, followed by calling IEBUPDTE to generate a PDS.

Example Code A3.1 Sample JCL for Running the AR2UPDTE Cataloged Procedure

```
//SAMPAR2U   JOB job card information
//*-----------------------------------------
--------------
//*  INVOKE AR2UPDTE FOLLOWED BY IEBUPDTE
//*
//*  REPLACE GENERIC DATA SET NAMES AS APPROPRIATE
//*-----------------------------------------
--------------
//STEP1        EXEC AR2UPDTE,OPTS='options'
//U2A.SYSARLIB DD DSN=input.ar370.archive,DISP=SHR
//U2A.SYSPUNCH DD DSN=updte.format.output,DISP=(OLD,PASS)
//STEP2        EXEC PGM=IEBUPDTE,COND=(0,NE),PARM='NEW'
//SYSPRINT     DD SYSOUT=*
//SYSUT2       DD DSN=output.pds,DISP=(NEW,CATLG),UNIT=SYSDA,
//             SPACE=(your-space-values),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=your-blksize)
//SYSIN        DD DSN=updte.format.output,DISP=OLD
//
```

In this example, *options* is any required AR2UPDTE options (for example, **'-t ?:QU'**).

The AR2UPDTE procedure contains the JCL shown in Example Code 18.2 on page 370.

Example Code A3.2 Expanded JCL for AR2UPDTE

```
//AR2UPDTE   PROC OPTS=''
//*************************************************************
//*  NAME:  AR2UPDTE                              (AR2UPDTE)   ***
//*  PRODUCT:  SAS/C                                           ***
//*  PROCEDURE:  CONVERT AR370 ARCHIVE TO IEBUPDTE INPUT       ***
//*  DOCUMENTATION:  SAS/C COMPILER AND LIBRARY USER'S GUIDE   ***
//*  FROM:  SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC     ***
//*************************************************************
//*
//U2A      EXEC PGM=AR2UPDTE,PARM='&OPTS DDN:SYSARLIB DDN:SYSPUNCH'
//STEPLIB  DD DSN=SASC.LOAD,
//            DISP=SHR        COMPILER LIBRARY
//         DD DSN=SASC.LINKLIB,
//            DISP=SHR       RUNTIME LIBRARY
//SYSTERM  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
```

Default Member Translation Rules

Unless the **-x** option is specified, some translations are automatically performed by the AR2UPDTE utility:

- If a period (.) is in a member name and it is not the first character, it is removed, and the rest of the member name is truncated (that is, MEMBER.NAME becomes MEMBER in the resulting IEBUPDTE file).
- If a period (.) is the first character of a member name, it is translated to an at sign (@).
- If blank ( )is the first character of a member name and it is not a translate character, then it is translated to a dollar sign ($).
- All member names are truncated to 8 characters since IEBUPDTE will not allow member names longer than 8.
- All member names are uppercased.

Note: Translations specified by the user occur prior to the default translations. Interactions between the user specified translations and the default translations may cause unexpected behavior. For example, if the **-t** option is invoked with **.:per**, then the default translation which converts a leading period (.) to the at sign (@) will not occur. The leading period (.) will be converted to "per". Also, if the **-t** option is invoked with **b:_**, then the b's will be converted to underscores (_) first and then to pound signs (#), by default. △

AR2UPDTE Diagnostics

The following diagnostic messages are generated by the AR2UPDTE utility. Diagnostic messages from the run-time library that further describe the problem may appear in conjunction with the AR2UPDTE diagnostics.

LSCAU1 Error: opening input file, "[*filename***]".**
An attempt to open the file *filename* failed. Under OS/390 this error occurs when the filename is a DDname and the DDname is not defined; but any file system problem or failure that might cause an open to fail could also cause this message.

LSCAU2 Error: opening output file, "[*filename***]".**
An attempt to open the file *filename* failed. Under OS/390 this error occurs when *filename* is a DDname and the DDname is not defined; but any file system problem or failure that might cause an open to fail could also cause this message.

LSCAU3 Error: reading file, "[*filename***]".**
An error occurred when attempting to read from the archive named *filename*. This diagnostic may be produced if the archive has been modified by any utility other than AR370 or UPDTE2AR; but any file system problem or failure that might cause a read to fail could also cause this message. Check all input files for validity and integrity.

LSCAU4 Error: writing file, "[*filename***]".**
An attempt to write one or more items to the output file stream has been unsuccessful. Usually this is caused by having insufficient space available for all the output; but any file system problem or failure that might cause a write to fail could also be the cause. Make sure the space available for the output file is large enough to hold all the output.

LSCAU5 Error: creating CMS-style file identifier from filename, "[*filename***] ".**
Files in CMS are named using a file identifier. The file identifier consists of three fields: filename, filetype, and filemode. An error occurred when attempting to create a valid CMS file identifier with the filename specified on the command line. All input and output files must have valid CMS-style file identifiers.

LSCAU6 Error: Wrong number of command line arguments.
Correct usage: ar2updte [-x | -t *c1*:*s1* [-t *c2*:*s2*...]]
filein fileout
The command line requires a minimum of two arguments, an input archive and an output filename.

LSCAU7 Error: loading list of translate characters.
Correct usage: ar2updte [-x | -t *c1*:*s1* [-t *c2*:*s2*...]]
filein fileout
The program failed while attempting to parse the options and translate characters in the command line. Be sure the command line is formatted correctly.

LSCAU9 Error: Option -t needs to be followed by an argument.
Correct usage: ar2updte [-x | -t *c1*:*s1* [-t *c2*:*s2*...]]
filein fileout
The **-t** option must be followed by an argument.

LSCAU10 Error: Unrecognized option -"*option***".**
Correct usage: ar2updte [-x | -t *c1*:*s1* [-t *c2*:*s2*...]]
filein fileout
The only valid options in AR2UPDTE are: **-x** and **-t** *c*:*s*.

LSCAU11 Error: The argument "argument" **that follows the -t option must be in the form *c*:*s* where *c* is the string to be translated and *s* is the resulting string.**
Correct usage: ar2updte [-x | -t *c1*:*s1* [-t *c2*:*s2*...]]
filein fileout
The **-t** option must be followed immediately with an argument in the form *c*:*s*. All strings *'c'* in the member names of the archive will then be translated to the string *'s'* in the resulting IEBUPDTE input file.

LSCAU12 Error: Unable to identify AR370 archive, "[*filename***]".**
An AR370 archive cannot be located from the *filename* specified in the command line. The input file specified on the command line must be a valid archive file.

LSCAU13 Error: reading AR370 archive members in "[*filename***]".**
An error occurred when attempting to read the members in the archive *filename*. This diagnostic may be produced if the archive has been modified by any utility other than AR370 or UPDTE2AR; but any file system problem or failure that might cause a read to fail could also cause this message. Check all input files for validity and integrity.

LSCAU14 Error: "[*filename***]" is not an AR370 archive.**
This file *filename* is not an archive. It cannot be processed as an archive. The input for AR2UPDTE must be an archive created by AR370 or UDPTE2AR.

LSCAU15 Error: File is not recognized as an archive. Cannot process file "[*filename***]".**
A file filename specified as an archive does not contain a valid archive header. Data read from the file is checked to verify that it is an archive. If the archive has been modified by any utility other than AR370 or UPDTE2AR, data could be lost or corrupted.

LSCAU16 Error: archive format unrecognized. Cannot process file "[*filename***] ".**
The file *filename* is an archive, but it contains an error in the symbol table. If the archive has been modified by any utility other than AR370 or UPDTE2AR, data could be lost or corrupted.

LSCAU17 Error: archive format unrecognized. Cannot process file "[*filename***] ".**
The file *filename* is an archive, but it contains an error in the string table. If the archive has been modified by any utility other than AR370 or UPDTE2AR, data could be lost or corrupted.

LSCAU18 Error: writing to output file, "[*filename***]".**
An attempt to write one or more items to the output file has been unsuccessful. Usually this is caused by having insufficient space available for all the output but any file system problem or failure that might cause a write to fail could also be the cause.

LSCAU28 Warning: The number of aliases for the member "[*member name***]" excee ds 16.**
The member *member name* is defined with more than 16 aliases. All of these aliases have been included in the resulting IEBUPDTE input format data file. However, IEBUPDTE cannot process members defined with more than 16 aliases. The excess alias cards should be removed before running IEBUPDTE.

LSCAU29 Warning: Duplicate member name "[*member name***]" has been generated i n output.**
member name is the identifier for more than one member in the archive. This name has been included more than once in the resulting IEBUPDTE input format file. However, the name of each PDS member must be unique, so before a partitioned data set is created, the IEBUPDTE input format file should be edited, or the archive should be manipulated using AR370 so that all members have unique names.

LSCAU30 Warning: Symbol "[*symbol name***]" was previously defined and has been omitted from output.**
Aliases are created for all symbols defined in each member of the archive. A symbol definition for *symbol name* appears in more than one member of the archive. Since PDS member and alias names must be unique, symbols that conflict with previous definitions have been omitted from the output. Linking

characteristics of the partitioned data set should still be preserved since only the first symbol defined by an archive is linked when using the archive.

UPDTE2AR Utility

The UPDTE2AR utility is a program that is used to create an AR370 archive by reading in the contents of a file in IEBUPDTE input format. The IEBUPDTE input file must contain 80-byte records in the format accepted by the OS/390 IEBUPDTE utility and described in the IBM manual MVS/DFP Utilities (SC26-4559). The file is divided into segments by IEBUPDTE "./ ADD" control records: each segment represents a single PDS member. A file can be generated in this format from an OS/390 card-image partitioned data set using the SAS System's SOURCE procedure. UPDTE2AR reads in this data and creates an AR370 archive. This archive can then be manipulated by the AR370 utility to delete, move, replace, view, or extract members.

UPDTE2AR options allow you to control the translation of PDS member names to archive member names. They also specify whether the archive's symbol table should mimic the source PDS directory or include all external symbols defined in members of the PDS.

Using UPDTE2AR under CMS

Under CMS, the UPDTE2AR utility is invoked directly with the following command:

```
updte2ar [options]  infile outfile
```

options specifies one or more options, each of which is a single character preceded by a hyphen (-). Some options (for example, **-t**) must be followed by an option argument. The argument may be separated from the option by white space, but this is not a requirement. Note that the case of option characters is not significant, but that case is significant for most option arguments.

The following options are recognized:

-a *ending* — appends the specified ending to the input member name to produce the output archive member name. The ending is limited to 8 characters.

-l — Converts the member names to lowercase.

-s — specifies that all external symbols defined in any input member are to be included in the archive symbol table. An archive produced with the **-s** option of UPDTE2AR has the same linking characteristics as an archive produced directly with AR370. If **-s** is omitted, then the archive symbol table references only the member names and aliases referenced by ./ control statements in the input file. An archive produced without **-s** has the linking characteristics of the source PDS.

-t *c*:*s* — specifies a translation rule to be used by UPDTE2AR when deriving an archive member name from a PDS member name. More than one **-t** option can be specified. The option argument *c*:*s* indicates that if the string *c* (which can be longer than a single character) occurs in an input member name, it is to be replaced by the string s in the output archive member name.

The *infile* and *outfile* arguments must be specified. The *infile* argument specifies the input file, which must be in valid IEBUPDTE input format. The *outfile* argument specifies the file identifier of the resulting output archive.

Note: Under CMS, the default filetype for the *infile* argument is IEBUPDTE. If filetype is specified, it must be joined to the filename with a period (.). The default filemode for the *infile* argument is *. The *outfile* argument specifies the file identifier of the resulting output archive. The default filetype for the *outfile* argument is A. If filetype is specified, it must be joined to the filename with a period (.). The default filemode for the *outfile* argument is *. △

The following examples show typical UPDTE2AR command lines on CMS.

`updte2ar test.iebupdte testlib.a`
Create a new archive named **`testlib.a`** using the IEBUPDTE input format file named **`test.iebupdte`**.

`updte2ar -t QU:?  -t $:x test3.iebupdte testlib3.a`
Create a new archive named **`testlib3.a`** using the IEBUPDTE input format file named **`test3.iebupdte`**. Convert all letters QU to question marks and then convert all dollar signs to x's.

`updte2ar -l -a .o test.iebupdte testlib4.a`
Create a new archive named **`testlib4.a`** using the IEBUPDTE input format file named **`test.iebupdte`**. Put all the member names in lowercase and append a **`.o`** to each member name. For example, the input member BUILD would be translated to the archive member **`build.o`**.

Using UPDTE2AR in TSO

This section describes calling UPDTE2AR in TSO using the TSO CALL command. Use the following syntax:

```
CALL 'library.name(UPDTE2AR)' '[options] infile outfile' ASIS
```

Here, *library.name* is the name of the library containing UPDTE2AR. Consult your SAS/C Software Representative for C compiler products for the correct library name.

See the section "Using UPDTE2AR under CMS" on page 373 for a description of the options. Note that the infile and outfile names will be interpreted as DDnames unless a SAS/C style prefix is used.

The following example shows a typical use of UPDTE2AR in TSO:

```
CALL 'SASC.LOAD(UPDTE2AR)' 'tso:testpds.data tso:testlib.a' ASIS
```

Using UPDTE2AR under OS/390 Batch

This section describes how to run UPDTE2AR under OS/390 batch using the UPDTE2AR cataloged procedure.

The UPDTE2AR cataloged procedure is used to execute UPDTE2AR under OS/390 batch. You must provide a SYSIN DD statement defining the IEBUPDTE format input file and a SYSARLIB DD statement defining the output AR370 archive. If you need to pass special UPDTE2AR options, specify the OPTS keyword in the EXEC statement, as shown below. See the section "Using UPDTE2AR under CMS" on page 373 for information on the UPDTE2AR options and their meanings.

Example Code 18.3 on page 375 shows typical JCL for using the UPDTE2AR cataloged procedure.

Example Code A3.3 Sample JCL for Running the UPDTE2AR Cataloged Procedure

```
//SAMPU2AR   JOB job card information
//*-----------------------------------------
--------------
//*  INVOKE UPDTE2AR
//*
//*  REPLACE GENERIC DATA SET NAMES AS APPROPRIATE
//*-----------------------------------------
--------------
//STEP1        EXEC UPDTE2AR,OPTS='options'
//U2A.SYSARLIB DD DSN=output.ar370.archive,DISP=OLD
//U2A.SYSIN    DD DSN=updte.format.input,DISP=SHR
//
```

In this example, *options* is any required UPDTE2AR options (for example, **'-t QU:?'**).

The UPDTE2AR contains the JCL shown in Example Code 18.4 on page 375.

Example Code A3.4 Expanded JCL for UPDTE2AR

```
//UPDTE2AR   PROC OPTS=''
//*************************************************************
//*  NAME:  UPDTE2AR                                (UPDTE2AR)   ***
//*  PRODUCT:  SAS/C                                             ***
//*  PROCEDURE:  CONVERT IEBUPDTE INPUT TO AR370 ARCHIVE         ***
//*  DOCUMENTATION:  SAS/C COMPILER AND LIBRARY USER'S GUIDE     ***
//*  FROM:  SAS INSTITUTE INC., SAS CAMPUS DRIVE, CARY, NC       ***
//*************************************************************
//*
//U2A      EXEC PGM=UPDTE2AR,PARM='&OPTS DDN:SYSIN DDN:SYSARLIB'
//STEPLIB  DD DSN=SASC.LOAD,
//            DISP=SHR        COMPILER LIBRARY
//         DD DSN=SASC.LINKLIB,
//            DISP=SHR       RUNTIME LIBRARY
//SYSTERM  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
```

UPDTE2AR Diagnostics

The following diagnostic messages are generated by the UPDTE2AR utility. Diagnostic messages from the run-time library that further describe the problem may appear in conjunction with the UPDTE2AR diagnostics.

LSCAU3 Error: Reading file, "[*filename***]".**
An error occurred when attempting to read from the input file *filename*. Check all input files for validity and integrity. Input files should be fixed-length record format with 80-byte records.

LSCAU4 Error: writing file, "[*filename***]".**
An attempt to write one or more items to the output file stream has been unsuccessful. Usually this is caused by having insufficient space available for all the output; but any file system problem or failure that might cause a write to fail

could also be the cause. Make sure the space available for the output file is large enough to hold all the output.

LSCAU5 Error: creating CMS-style file identifier from filename, "[*filename*] ;".
Files under CMS are named using a file identifier. The file identifier consists of three fields: filename, filetype, and filemode. An error occurred when attempting to create a valid CMS file identifier with the filename specified on the command line. All input and out put files must have valid CMS-style file identifiers.

LSCAU6 Error: Wrong number of command-line arguments.
Correct usage: updte2ar [-l] [-s] [-a *ending*] [-t *c1*:*s1* [-t *c2*:*s2*...]] *filein fileout*
The command line requires a minimum of two arguments, an input archive and an output filename.

LSCAU7 Error: loading list of translate characters.
Correct usage: updte2ar [-l] [-s] [-a *ending*] [-t *c1*:*s1* [-t *c2*:*s2*...]] *filein fileout*
The program failed while attempting to parse the options and translate characters specified on the command line. Be sure the command line is formatted correctly.

LSCAU8 Error: Argument following -a cannot be longer than 8 characters.
Correct usage: updte2ar [-l] [-s] [-a *ending*] [-t *c1*:*s1* [-t *c2*:*s2*...]] *file-in file-out*
The **-a** option specified a suffix that was more than 8 characters.

LSCAU10 Error: Unrecognized option -*option* .
Correct usage: updte2ar [-l] [-s] [-a *ending*] [-t *c1*:*s1* [-t *c2*:*s2*...]] *filein fileout*
The only valid options in UPDTE2AR are: **-l**, **-s**, **-a** *ending*, **-t** *c*:*s*.

LSCAU11 Error: The argument "argument" **that follows the -t option must be in the form *c*:*s* where *c* is the string to be translated and *s* is the resulting string.**
Correct usage: updte2ar [-l] [-s] [-a *ending*] [-t *c1*:*s1* [-t *c2*:*s2*...]] *filein fileout*
The **-t** option must be followed immediately with an argument in the form *c*:*s*. All strings *c* in the member names of the IEBUPDTE file will then be translated to the string *s* in the resulting AR370 archive.

LSCAU19 Error: invalid name for symbol, "*symbolname*" specified in a SYMDEF control st atement.
SYMDEF symbols must be 1 to 8 characters in length. The symbol name *symbolname* is too long. Symbols specified via SYMDEF control statements must be at least 1 character and not more than 8 characters in length. Check the SYMDEF cards in the input object files.

LSCAU20 Error: invalid SYMDEF control card in file "*filename*".
An AR370 SYMDEF control statement in the input file *filename* contained invalid syntax. Check the SYMDEF control statement in the specified input file to make sure it conforms to the general form and syntax of linkage editor control statements. Make sure the symbol names are between 1 and 8 characters in length.

LSCAU21 Error: Unable to write object to AR370 archive file, "*filename*"**.**

An attempt to write one or more items to the output file stream has been unsuccessful. Usually this is caused by having insufficient space available for all the output; but any file system problem or failure that might cause a write to fail could also be the cause. Make sure the space available for the output file is large enough to hold all the output.

LSCAU22 Error: Encountered EOF in continued SYMDEF card in file, "*filename*"**.**

An AR370 SYMDEF control statement in the file *filename* is invalid. An End of File was encountered in place of the continuation of the SYMDEF card. Check the SYMDEF cards in the input file.

LSCAU23 Error: Unable to open IEBUPDTE file, "*filename*"**.**

An attempt to open the file *filename* failed. Under OS/390, this error occurs when the *filename* is a DDname and the DDname is not defined; but any file system problem or failure that might cause an open to fail could also cause this message.

LSCAU24 Error: Unable to open AR370 archive file, "*filename*"**.**

An attempt to open the file *filename* failed. Under OS/390, this error occurs when the *filename* is a DDname and the DDname is not defined, but any file system problem or failure that might cause an open to fail could also cause this message.

LSCAU25 Error: Read of input file, "[*filename***]" failed.**

When attempting to read the input file named filename, UPDTE2AR was unable to read 80 bytes. The IEBUPDTE utility requires the input file to be 80-byte records, blocked or unblocked. Check the input file for validity and integrity.

LSCAU26 Error writing library header to output file, "*filename*"**.**

An attempt to write one or more items to the output file stream has been unsuccessful. Usually this is caused by having insufficient space available for all the output, but any file system problem or failure that might cause a write to fail could also cause this message. Make sure the space available for the output file is large enough to hold all the output.

LSCAU27 Error in seeking to offset in file, "*filename*"**.**

An error occurred when attempting to position to an offset in the file *filename*.

APPENDIX

4

The CMS GENCSEG Utility

Introduction

GENCSEG is a utility program for VM and CMS users that installs LOADLIB members in a Discontiguous Saved Segment (DCSS), hereafter referred to as a segment. Once installed, these members may be dynamically loaded by using the **`addsrch`** and **`loadm`** functions.

There are several reasons for using the GENCSEG utility. For example, programs that reside in a segment can be attached outside of the virtual machine's address range and, therefore, do not occupy memory within the virtual machine. Also, several users can share a copy of the segment. Further, since dynamic loading from a segment uses CP to fetch load modules, this type of installation is useful for programs that rely heavily on dynamic loading.

The GENCSEG utility operates as follows. First, GENCSEG accumulates a list of LOADLIB members. The next step is to determine the address range of the segment. Next, GENCSEG allocates a directory space and then begins to load the members in sequential order. As the members are loaded, their names are added to the directory. By default, the members are added sequentially, but GENCSEG provides parameters to specify load order, alignment, and spacing. The load address for the first member

always begins immediately following the directory. The load address is then updated according to requested alignment and other options.

How Dynamic Loading Uses a Segment

The **loadm** function is used to dynamically load external load modules. Under CMS, **loadm** searches for the load module in several locations. The search is controlled by the **addsrch** function, which defines a set of locations where dynamically loaded modules can be found. The CMS_DCSS argument to **addsrch** specifies that modules can be found in a segment that has been created by GENCSEG.

The Segment Installation Process

A user with CP class E privileges can create and maintain new segments. Once a segment is defined or available, you can use GENCSEG to install members of one or more LOADLIBs into a single segment. The installation can be controlled by GENCSEG parameters that specify which members are to be installed, the names of the LOADLIBs to be used, and the order in which the members are to be installed.

GENCSEG always loads the members in your virtual machine, creating an image of the segment to be saved. This means that GENCSEG must be invoked in a virtual machine that has enough virtual storage defined to contain the entire segment, plus the memory required by GENCSEG to operate and the memory required by CMS. Before installing any members, GENCSEG initializes the segment image by writing binary 0s into the entire image area.

GENCSEG always assumes that the segment is designated as shared and protected under VM/SP. In a VM extended architecture system, this implies that the SR parameter is used in the DEFSEG command. After the image area has been set to 0, GENCSEG performs the equivalent of the SETKEY command, setting all storage keys in the segment image to key 0. This ensures that the code and data in the segment do not become corrupted during use. Since the code and data will reside in protected memory, the code must be reentrant. Non-reentrant code in a shared segment causes protection exceptions when executed. Use the SAS/C Compiler option **rent** to allow reentrant modification of static and external variables. See Chapter 6, “Compiler Options,” on page 101.

The **loadm** function searches for modules in a segment by reading a directory in the segment itself. This directory is created by GENCSEG during the installation process and begins at the first address in the segment. There is one entry in the directory for each member or alias installed in the segment.

GENCSEG saves the segment name and the current date and time in the first 16 bytes of the directory. The directory is terminated by an entry containing all binary 1s. Each directory entry is four words long. The first two words contain the name of the member or alias. The third word contains the address of the module entry point. The fourth word is all binary 0s and is reserved for use by the library.

The GENCSEG load parameters LOADLIB, ALIGN, PAGE, and SPACE control the installation process. These parameters are processed from left to right in the order they are entered on the command line. Upon invocation, GENCSEG initializes the segment and then begins processing the load parameters.

As members are loaded, GENCSEG maintains the value of the next available address in the segment. This value, called the load address or loadaddr, has an initial value equal to the first address following the segment directory. After each member is loaded, *loadaddr* is incremented by the size of the member. The ALIGN, PAGE, and SPACE parameters can be used to modify the value of *loadaddr*. However, *loadaddr* cannot be decremented, and at no time can the value of *loadaddr* become larger than

the last address defined in the segment. If this occurs, GENCSEG terminates with a diagnostic message, and the segment is not saved.

GENCSEG installs a LOADLIB member by relocating the code and data based on loadaddr and creating a segment directory entry for the member. This process is repeated until all the members specified are installed. GENCSEG then issues a SAVESYS command for the segment.

As the installation proceeds, GENCSEG types a variety of diagnostic messages to the terminal. These messages report on the parameters being used, the state of the virtual machine, where the members are being loaded, and how many pages are used. GENCSEG also issues diagnostics when something unexpected occurs or when it is unable to continue the installation.

When all of the load parameters are processed (and no errors have occurred), GENCSEG invokes the CP SAVESYS (under VM/SP) or SAVESEG (under VM/XA or VM/ESA) command to save the segment image into DASD storage. The SAVESYS command must be issued from a userid with a CP privilege class of E. If the userid does not have CP class E privileges, the command fails and the segment is not saved.

GENCSEG Parameters

Table A4.1 on page 381 lists all GENCSEG parameters. A description of each parameter follows the table. In the Type column, an L indicates load parameter, an O indicates option parameter.

Table A4.1 GENCSEG Parameters

Parameter Name	Format	Type	Explanation of Format
segment *name*	*name*	L	*name* is the name of the segment to be created.
LOADLIB	—1*name*[(*mem1*[,*mem2*...])]	L	name is the name of the LOADLIB to be installed; mem1, mem2, and so on are optional LOADLIB members.
ALIGN	-a<*n*>	L	*n* is an integer.
PAGE	-p<*n*>	L	*n* is an integer.
SPACE	-s<*n*>	L	*n* is an integer.
EDIT	-e	O	See description.
JAPAN	-j	O	See description.
LOADALL	-r	O	See description.

Load Parameters

With the exception of the segment name parameter, all GENCSEG parameters are designated by a hyphen (-) followed by a letter. For example, -a designates the ALIGN parameter. In general, use of any single load parameter causes all of the default load parameters to be overridden.

The Segment Name Parameter

The name of the segment to be created must appear as the first parameter on the command line. Do not prefix the name with a hyphen. The name must match the name used in the segment definition. Specifying a segment name does not override any other default load parameters. For example, the following command indicates that GENCSEG should create a segment named MYDCSS:

```
GENCSEG MYDCSS
```

CAUTION:

Do not invoke GENCSEG without entering a segment name and the name of at least one LOADLIB following the LOADLIB (-l) option. By default, if you do not specify a segment name and the name of a LOADLIB, GENCSEG attempts to install the compiler and run-time library into the LSCRTL segment. △

The LOADLIB Parameter: -l

The LOADLIB parameter (-l) specifies the name of a LOADLIB file from which members are to be installed. For example, the following causes GENCSEG to install all the members of PROJECT LOADLIB:

```
GENCSEG MYDCSS -lPROJECT
```

GENCSEG searches for PROJECT LOADLIB on any ACCESSed minidisk, using the normal CMS search order. All of the members in PROJECT LOADLIB are installed in the order they are found in the LOADLIB directory, beginning at the address pointed to by the load address.

If you do not want all the members of the LOADLIB to be installed or if you want to specify a different order, append a list of member names to the LOADLIB filename. The member name list must be enclosed in parentheses, and member names must be separated with a comma. For example, the following command causes GENCSEG to install MEM1 first, followed by MEM2 and MEM4:

```
GENCSEG MYDCSS -lPROJECT(MEM1,MEM2,MEM4)
```

This command causes GENCSEG to install MEM4 first, followed by MEM2 and MEM1:

```
GENCSEG MYDCSS -lPROJECT(MEM4,MEM2,MEM1)
```

Any other members in PROJECT LOADLIB are ignored.

Alias entries in LOADLIBs

LOADLIB directories can have alias entries. Aliases are created by the LKED command and are used to specify alternative names and possibly alternative entry points to a member known as the parent member. GENCSEG does not install a new copy of the parent for an alias name. Instead, it adds an entry in the segment directory for the alias name, where the entry point given in the segment directory refers to a location in the installed parent member. You can specify an alias name in a member name list either before or after the occurrence of the parent member name. GENCSEG creates the directory accordingly.

However, if you specify an alias name in the member name list, you must also specify the name of the parent member. If GENCSEG encounters an alias name and does not find the corresponding parent name, the installation is terminated.

The ALIGN Parameter: -a

Use the ALIGN parameter (-a) to indicate that *loadaddr* should be increased to the next address that is a multiple of *n* K, where n is an integer immediately following the parameter. *n* can be 2, 4, 8, or 64, as in the following example:

```
GENCSEG MYDCSS -lPROJECT -a2 -lPROJECT2
```

In this example, all the members of LOADLIB PROJECT are installed starting at the next address after the end of the directory. *loadaddr* is then aligned to the next 2K address, and the members of LOADLIB PROJECT2 are installed. If ALIGN is not specified, then *loadaddr* is the next available address.

The PAGE Parameter: -p

Use the PAGE parameter (-p) to cause members to be aligned to a page offset from the beginning of the segment. The following example requests alignment to the beginning of the fifteenth 4K page from the beginning of the segment:

```
GENCSEG MYDCSS -p15
```

Each page of memory requested by the PAGE parameter equals 4K bytes. Note that the PAGE option cannot make loadaddr point to a lower address.

The SPACE Parameter: -s

Use the SPACE parameter (-s) to add an *n* K value to *loadaddr*, where *n* can be between 1 and 256 inclusive. The following example indicates that loadaddr should be incremented by 1K after installing LOADLIB PROJECT and before installing PROJECT2:

```
GENCSEG MYDCSS -lPROJECT -s1 -lPROJECT2
```

If you do not specify SPACE, *loadaddr* is incremented only by the size of the member being loaded.

Calculating the Total Segment Size

Upon completion, GENCSEG reports the total size of the segment in bytes. This value includes the total size of the directory, the members installed, and the amount added by the SPACE, PAGE, and ALIGN parameters.

Option Parameters

GENCSEG has three option parameters, EDIT, JAPAN, and LOADALL. Use of these options does not override the default load parameters.

The EDIT Option: -e

The EDIT option (-e) allows trial invocations of GENCSEG. You can use the EDIT option to see what happens when various combinations of load parameters are used

without actually having to load and save the segment. The following example indicates that the installation of the members of PROJECT and PROJECT2 LOADLIBs is only to be simulated:

```
GENCSEG MYDCSS -e -lPROJECT -s1 -lPROJECT2
```

When the EDIT option is in effect, you can specify the name of a segment that has not yet been defined. GENCSEG initializes *loadaddr* to 0 and allows it to be as large as necessary to complete the segment creation.

The EDIT option also suppresses initialization of the image to binary 0s and storage key initialization. GENCSEG does not check the virtual machine size. Relocation of code and data is suppressed. Of course, the SAVESYS or SAVESEG command is not issued.

The JAPAN Option: -j

If the JAPAN option (-j) is used, GENCSEG types all diagnostic messages in uppercase. This option is intended for use with printers and terminals that have only uppercase roman characters.

The LOADALL Option: -r

The LOADALL option (-r) forces GENCSEG to attempt to load all of the members of a specified LOADLIB without regard to the location of the segment or to the RMODE of a member. The LOADALL option causes GENCSEG to behave as it did in releases prior to 5.50C.

If you specify LOADALL, then, as the list of LOADLIB members is made, GENCSEG inspects each member's RMODE and compares that to the location of the segment. If the LOADALL option has been specified, then GENCSEG attempts to load every member in the list. An error will occur if the segment has been defined to have an ending address greater than 16 megabytes and at least one member in the list is found to have RMODE=24.

If the LOADALL option is not specified, then GENCSEG will do the following:

- If the ending address of the segment is greater than 16 megabytes, only members with RMODE=ANY are loaded.
- If the ending address of the segment is less than 16 megabytes, only members with RMODE=24 are loaded.

The GENCSEG Listing File

As it executes, GENCSEG creates a file on your A-disk called GENCSEG LISTING. This file contains a copy of the diagnostic messages typed on the terminal. GENCSEG never erases the current copy of GENCSEG LISTING; instead, it appends the new output to the end of any existing copy.

Virtual Machine Requirements when Using GENCSEG to Save a Segment

When GENCSEG is invoked to create and save the segment, it must be executed in a virtual machine with CP privilege class E so that the user can issue the SAVESYS or SAVESEG command. The virtual machine also must have enough storage defined to

contain the segment image, and it must have the storage required by CMS and GENCSEG itself.

Renaming the Default Run-Time Library Segment

One method of installing a program in a segment is to include a copy of the SAS/C transient run-time library LOADLIB in the same segment. You can, for example, invoke GENCSEG with the following load parameters:

```
GENCSEG MYDCSS -lLSCRTL -lPROJECT
```

Since all C programs use the name LSCRTL as the default name of the run-time segment, you need to modify the default value to use MYDCSS instead. All C main load modules (that is, those with a **`main`** function) include a CSECT that contains the default run-time library name LSCRTL at offset 0. The CSECT name is LCSEGC. Use the CMS ZAP command to alter the default name in the main load module. Both the old (VER) name and the new (REP) name must be uppercase, left-adjusted, and padded with blanks to eight characters. For example, if the main load module is in MYPROG MODULE, the following zap changes the default segment name from LSCRTL to MYDCSS:

```
NAME MYPROG L$C$SEGC
VER  00 D3E2C3D9E3D34040  "LSCRTL  "
REP  00 D4E8C4C3E2E24040  "MYDCSS  "
```

Note: Data in quotes are not to be included in the zap. △

After the zap is applied, MYPROG MODULE and any load modules loaded by it will use the run-time library in MYDCSS.

Sample GENCSEG Listings

The sample GENCSEG listing in Example Code 19.1 on page 386 shows the output produced by GENCSEG under these conditions:

- GENCSEG is executing under VM/ESA.
- The userid invoking the program is SASCUSER. Two LOADLIBs are involved, both of which are on minidisk labeled PROJEC.
- The segment named MYDCSSA is defined with the following **`defseg`** command, which causes the segment to attach at virtual address '1000000'X:

  ```
  DEFSEG MYDCSSA 1100-11FF SR
  ```

- The highest address defined in MYDCSSA is '11FFFFF'X.
- The LOADALL(-r) option is not specified, so only members with RMODE=ANY are loaded.

The GENCSEG command line in Example Code 19.1 on page 386 has been entered as follows:

```
GENCSEG MYPROJB -e -lPROJECT -lPROJECT2(PHASE3,PHASE5,PHASE6)
```

PROJECT LOADLIB has three members, PHASE1, PHASE2, and PHASE4, plus an alias for PHASE1 named STARTUP. Only members PHASE3, PHASE5, and PHASE6

are to be installed from PROJECT2 LOADLIB. Since the LOADALL option was not specified, only the members with RMODE=24 are loaded.

Example Code A4.1 GENCSEG Listing

```
GENCSEG Release 6.00C                                              Page 1
Copyright(c) 1996 by SAS Institute Inc., Cary, NC USA

   LSCG022 Note: GENCSEG invoked by userid: SASCUSER

   LSCG013 Note: Using 'MYPROJA ' as the segment name.

   LSCG035 Note: 'MYPROJA ' segment loads from address '1100000'X
                 to address '011FFFFF'X.

   LSCG043 Note: Edit mode.

   LSCG071 Note: Only members having an RMODE = ANY will be
           loaded.

   LSCG069 Note: The LOADALL (-r) option may be used to load all
                 members of a LOADLIB (if possible) regardless of a
                 members RMODE.

   LSCG070 Note: 7 total members/aliases processed.

   LSCG037 Note: 3 members/aliases to be loaded.

   LSCG038 Note: Directory size: '50'X (80) bytes.

Wed May 06 10:40:40 1992          --- MYPROJA ---                  Page 2


               Loading from: PROJECT LOADLIB A1 on PROJEC

Member/Alias   Entry Point   Origin     Size    Alias of  Loaded? RMODE
  PHASE1        01100264    01100050  0000B0E8              Y      ANY
  STARTUP       01100264    01100050  0000B0E8   PHASE1     Y      ANY
  PHASE2        ********    ********  ********              N      24
  PHASE4        ********    ********  ********              N      24


Wed May 06 10:40:40 1992          --- MYPROJA ---                  Page 3


               Loading from: PROJECT2 LOADLIB A1 on PROJEC

Member/Alias   Entry Point   Origin     Size    Alias of  Loaded? RMODE
  PHASE3        ********    ********  ********              N      24
  PHASE5        ********    ********  ********              N      24
  PHASE6        0110B34C    0110B138  0000B0E8              Y      ANY


Wed May 06 10:40:40 1992          --- MYPROJA ---                  Page 4
```

```
LSCG050 Note: Total size (including directory): '16220'X bytes.

LSCG053 Note: 23 page(s) used.  Hexadecimal page number range:
              1100-1117.

LSCG039 Note: Processing completed.
```

APPENDIX

5

Sharing extern Variables among Load Modules

Global extern Variables

If the **rent** or **rentext** compiler option is used, the compiler generates an object code data type called a pseudoregister for each **extern** variable. If the **norent** compiler option is used, the compiler generates a pseudoregister only for **extern** variables declared as **__rent** or whose names begin with an underscore (such as **_options**). When the program is linked, the linker assigns an offset to each pseudoregister and accumulates the total length. At execution time, storage is dynamically allocated for all the pseudoregisters in the load module. This storage is known as the pseudoregister vector.

In a program consisting of several load modules, each load module has its own pseudoregister vector that is independent of the pseudoregister vector of any other load module. This means that, in general, **extern** variables cannot be shared between load modules.

However, certain **extern** variables need to be available in all the load modules in a multi-load-module program. An **extern** variable of this sort is known as a global **extern**. During dynamic loading, the **loadm** function copies the values of the global extern from the pseudoregister vector of the main load module to the psuedoregister vector of the newly loaded module. Of course, **loadm** must be able to determine the length and offset of each global **extern** both in the calling and called load modules. This copy takes place only when the load module is loaded. Changes in the value of the global **extern** after that point are not reflected back to any other load module.

L$UGLBL

L$UGLBL is an assembler language program that contains the list of global **extern** variables. The object code generated by L$UGLBL must be linked with every C load module. By default, this list contains only those global **extern** variables that are required by the library, such as a pointer to **errno**. Table A5.1 on page 390 is a list of global **extern** variables that are defined by default.

All default global **extern** variables are defined as pointers. This allows a single copy of the variable to be defined in the primary load module. Because references to its value are always made via the pointer, all load modules that refer to the global **extern** refer to the current value. For example, the global **extern __en** points to an **extern** named

errno in the primary load module. In **<errno.h>**, **errno** is a macro defined as **(*__en)**. Therefore, any reference to **errno** will always refer to **errno** in the primary load module.

Note: Because the compiler changes underscores in external identifiers to pound signs (#), the # characters in these names correspond to underscores in the C identifier. For example, ##EN in L$UGLBL corresponds to **__en** in **<errno.h>**. △

Table A5.1 Default Global extern Variables

Name	Points to
##IO	global I/O information
##CT	**ctype** table
##EN	**errno**
##MN	**_msgno**
##MG	miscellaneous information
#ENVIRON	POSIX environment pointer address

For CICS applications only, there are three additional global **extern** variables in the L$UGLBL list. They are listed in Table A5.2 on page 390. These additional global **extern** variables appear immediately after the default global **extern** variables in the L$UGLBL list.

Table A5.2 Default CICS Global extern Variables

Name	Points to
#EIBPTR	CICS EXEC interface block
#COMMPTR	CICS COMMAREA
#DOBPTR	DL/I interface block

Another global **extern** variable has been added for users of the Generalized Operating System (GOS) feature. This new **extern** variable appears after the other default global **extern** variables; it only exists for users of GOS. The name of this **extern** is ##GA. ##GA points to the GOS anchor block.

The list in L$UGLBL can be extended to include those variables that are required by a specific site or application. The GLOBAL assembler macro, defined in L$UGLBL, is used to define additional entries. This macro adds a pseudoregister to the L$UGLBL list. This is the format of the GLOBAL macro instruction:

```
GLOBAL name,CUST=(YES|NO)
```

name is the name of the global **extern**. If the C identifier contains underscores, the corresponding characters in *name* must be pound signs (**#**).

CUST=YES indicates that *name* is a custom addition to the list. All additions to the global list must use this operand.

CUST=NO is reserved. This is the default.

For example, use the following code to add a global **extern** named **gcb**:

```
GLOBAL GCB,CUST=YES
```

All custom global **extern** variables must be defined at the end of the default list. Global **extern** variables are aligned on a doubleword boundary. Each global **extern** must be no more than 4 bytes long.

L$UGLBL is in SASC.SOURCE(L$UGLBL) under OS/390 and is member L$UGLBL of LSU MACLIB under CMS. Consult your SAS Software Representative for C compiler products for more information.

Cautions

The following cautions should be noted in sharing **extern** variables among load modules:

- To ensure that the compiler creates a pseudoregister for global **extern** variables even if the **norent** option is in effect, declare the variable **__rent**, or use an underscore as the first character of the identifier.
- Names of **extern** variables that differ only in case are treated as identical. **extern** identifiers that differ only after the first eight characters are treated as identical.
- External identifiers beginning with two underscores or a single underscore followed by an uppercase character are reserved for the implementation, according to the ANSI Standard.
- The default list must not be changed, either in number or order. Such a change prevents the library from operating correctly and will probably cause an abend.
- The default list may change from release to release as necessary.
- If the L$UGLBL list is modified, all non-C-library load modules that have been linked with the list must be relinked with the modified version.
- L$UGLBL is not supported in SPE.

For more information about external identifiers, refer to “External Variables” on page 55.

APPENDIX

6

Using the indep Option for Interlanguage Communication

Introduction

This appendix explains how to use the **indep** compiler option to generate code that can be used in both simple and complex multilanguage programs.

The **indep** compiler option has two quite distinct uses. The first is to allow C code to be called directly from other high-level languages. The second is to avoid the repeated creation and destruction of the C framework when the SPE feature is used. This appendix describes the first use of the **indep** option. The second use is described in Chapter 14, "Systems Programming with the SAS/C Compiler," on page 273.

Unless your multilanguage application is simple, you need to understand execution frameworks in order to use the **indep** option for interlanguage communication. This appendix describes what an execution framework is and then describes how to use **indep** to create the C execution framework and to switch between frameworks. A complete example, which uses **indep** to switch between the C and the PL/I frameworks, follows the description.

Most interlanguage applications use the normal C framework, which supports the entire SAS/C run-time library. In many cases, the techniques described in this appendix also can be used with the minimal SPE framework described in Chapter 14, "Systems

Programming with the SAS/C Compiler," on page 273. SPE is explicitly mentioned here only for exceptional cases where the SPE behavior differs from the full library behavior.

It is recommended that new multilanguage applications use the facilities described in the SAS/C Compiler Interlanguage Communication Feature User's Guide, rather than using **indep** directly, as described here. In most cases, these facilities are easier to use than using **indep** and are equally powerful.

Simple Multilanguage Programs

You can use the **indep** option for simple multilanguage programs without reading the rest of this appendix. Simple multilanguage programs are programs in which the other language calls C functions but C never calls the other language. (Even simpler types of multilanguage programs are discussed in Chapter 12, "Simple Interlanguage Communication," on page 227.)

For this type of program, compile all of the C functions that can be called from the other language with **indep**, and link-edit normally. A non-reentrant load module is produced. If reentrancy is required, you need to read the description of the L$UPREP exit routine later in this appendix. Also, in order to close files and free memory allocated by the C program, you may want to call the library routine L$UEXIT (CEXIT in COBOL or FORTRAN) from the other language after all calls to C have completed. (L$UEXIT is described in more detail later in this appendix.)

If you plan to pass arguments to C functions, also read "Calls to C" on page 400.

Execution Frameworks

In general, successful execution of code written in a high-level language requires the accessibility of an appropriate execution framework. An execution framework (also called an environment) is a collection of data and routines supporting the execution of code. (For example, memory allocation tables and error-handling routines are frequently components of an execution framework.) Note that for code to execute successfully, it is not sufficient for the execution framework to exist; it also must be accessible and available to the program. This means, in general, that machine registers must be set up to address components of the framework.

Since each language has its own conventions for access to its framework, it is usually impossible for more than one framework to be accessible at once. Therefore, a call from one language to another must switch frameworks, that is, make the new language's framework accessible (or active) before performing the call and restore the calling framework after the call is complete.

The C Execution Framework

The C execution framework includes the stack from which save areas and auto storage are allocated, the pseudoregister vector that contains **extern** and **static** data for reentrant programs, and the C Run-Time Anchor Block (CRAB). The CRAB contains constants and other information used by both compiled code and the library. The C framework also includes the default signal handlers set up by the library. When the C framework is active, register 12 addresses the CRAB. C functions compiled without the **indep** option expect register 12 to address the CRAB on entry; if it does not, the results are unpredictable.

When a C program compiled without the **indep** option is executed, the C framework is created by the L$CMAIN library routine. The execution of this routine precedes execution of the **main** function. L$CMAIN obtains storage for the CRAB, the pseudoregister vector, and the initial stack and heap and sets register 12 to address the CRAB.

C Execution Framework Creation with indep

The C framework is created when the first (or initial) C function is called. (This function is not necessarily named **main**.) Because this function is called before the C framework is created, it must be compiled with **indep**. (See "main Function Considerations" on page 400 for one exception.)

When a C source file is compiled with the **indep** option, all the functions in the resulting object file have the following properties:

- The function can be entered before the C execution framework has been created.
- The function does not depend on the contents of register 12 on entry.
- On entry, the compiled code for the function invokes an exit routine, L$UPREP, to make the C execution framework available.
- The function expects to be called with standard IBM 370 linkage (that is, with register 13 addressing a 72-byte save area).

Therefore, a function compiled by **indep** can be called directly from outside of the C execution framework (for example, with another high-level language's framework active).

The primary characteristic of the object code generated when the **indep** compiler option is used is that the L$UPREP routine is invoked whenever a function compiled with the **indep** option is called. L$UPREP determines whether the C execution framework has been created. If the C framework has been created, the CRAB address is loaded into register 12 and execution of the called function proceeds normally. If not, L$UPREP invokes L$UMAIN, an initialization routine in the run-time library, to create the C framework. After the C framework is created (and the CRAB address is stored in register 12), execution of the initial function proceeds normally.

L$UPREP is provided in source code on the SAS/C installation tape. Therefore, you can modify its check for the existence of the C execution framework as necessary to suit your application. L$UPREP uses an assembler macro named L$UCENV to locate a word in which the CRAB pointer is to be saved for future reference. The L$UCENV macro sets R12 to the address of this word. As provided, L$UCENV defines a CSECT named L$UCENV to be used to hold the CRAB address.

Because all functions compiled with **indep** invoke L$UPREP, it does not matter which of these functions is called first. Whichever function is called first creates the C framework.

Specifying Run-Time Library Options

Normally, when a C program is invoked by the operating system, run-time library options are specified as part of the argument list passed by the operating system. However, when the initial function is compiled with the **indep** option, the arguments can have any type and therefore cannot be modified, or even inspected, by the library. For this reason, another mechanism must be used in such an application to specify run-time options.

One method is the normal technique of initializing external **int** variables named **_options**and **_negopts** and the external **char** variable named **_linkage** to specify the options required. (Refer to Chapter 9, "Run-Time Argument Processing," on page 185

for more details.) However, this technique can be used only when the options required are constant and known at compile time.

To support varying options (that is, options that can vary between executions), you can provide a routine named L$URTOP. This is an optional routine. If L$URTOP is not included in the load module, only default options (or those specified by external variables) are used. If L$URTOP is present in the load module, it is called during the initialization of the C framework, using standard linkage. No arguments are passed to it, and it should return the address of a run-time options string in register 15. The string should contain a halfword length field, followed by the required options. Note that the length contains the number of characters in the string, not the number of characters plus two. During later initialization processing, the string is tokenized exactly as if it came from a normal command line. However, if any of the tokens in the string are not run-time library options, diagnostic messages are generated.

Because SPE does not support run-time options, L$URTOP is not used with the SPE framework.

The following example shows a typical L$URTOP routine:

Example Code A6.1 Sample L$URTOP Routine

```
*********************************************************************
*                                                                   *
* L$URTOP is an optional routine to provide runtime arguments to    *
* Indep applications.                                               *
*                                                                   *
* In this example, the normal action is to specify a minimal set of *
* run-time options.  However by zapping the location DEBUGOPT, two  *
* completely different sets of options can be provided.             *
*                                                                   *
* When DEBUGOPT is set to 1 a different set options more suitable   *
* for limited debugging is specified. This set of options also      *
* define environment variables (APL1) meaningful to the application.*
* Additionally, the runtime options at label ALTOPTST can be zapped *
* to provide an entirely different set of runtime options when      *
* DEBUGOPT is set to 1.                                             *
*                                                                   *
* When DEBUGOPT is set to 2 a different set of options provide the  *
* information necessary to start the SAS/C Remote Debugger.         *
*                                                                   *
*********************************************************************
L$URTOP  CSECT
         SPACE
*********************************************************************
* Ensure AMODE/RMODE match those of the SAS/C Library Default       *
*********************************************************************
L$URTOP  RMODE   ANY
L$URTOP  AMODE   31
         SPACE
         USING *,15              Tell Assembler
         B     SETARGS           Branch around eye-catcher
         DC    AL1(L'EYECATCH)
EYECATCH DC    C' L$URTOP - Sample'
         DC    XL1'00'            Filler
         SPACE
* Switch to determine which set of runtime options will be used
         SPACE
```

```
DEBUGOPT DC    XL1'00'
         SPACE
SETARGS  DS    0H
         STM   14,12,12(13)       Save entry regs in callers area
         SPACE
         LA    1,STDOPTS          Default to standard runtime
         CLI   DEBUGOPT,0         Has switch been set for alternate?
         BE    RETARGS            No,  return standard runtime
         SPACE
* Alternate has been requested, set as indicated or take the default
CHKALT   DS    0H                 Check for Alternate runtime
         CLI   DEBUGOPT,1         Has switch been set for alternate?
         BNE   CHKRMT             No,  check for Remote Debugger
         LA    1,ALTOPTS          Yes, return optional runtime
         B     RETARGS
CHKRMT   DS    0H                 Check for Remote Debugger Setup
         CLI   DEBUGOPT,2         Has switch been set for Debug Rmt?
         BNE   RETARGS            No,  return standard runtime
         LA    1,DBGRMT           Yes, return Debugger Remote runtime
         SPACE
RETARGS  LR    15,1               Point R15 at runtime arguments
         DROP  15
         SPACE
         L     14,12(,13)         Restore R14
         LM    0,12,20(13)        Restore other registers
         BR    14                 Return to caller
         LTORG ,
*********************************************************************
* Minimal Options                                                   *
*********************************************************************
STDOPTS  DC    AL2(L'STDOPTST)
STDOPTST DC    C'=VERSION'        Standard - runtime
         SPACE
*********************************************************************
* Alternate options providing additional information                *
*********************************************************************
ALTOPTS  DC    AL2(L'ALTOPTST)    Alternate- runtime
ALTOPTST DC    CL80'=BTRACE =WARNING =FDUMP =APL1=DEBUG =STORAGE'
         DC    0D'0'              Filler
         SPACE
*********************************************************************
* Alternate options which provide setup information for the SAS/C    *
* Remote Debugger.                                                  *
*********************************************************************
DBGRMT   DC    AL2(DBGLEN)          RmtDebug - runtime
RMTOPTS  DC    C'=_DB_COMM=TCPIP '
         DC    C'=_DB_HOST=124.383.1.2 '
         DC    C'=_DB_PORT=3123 '
         DC    CL80'=DEBUG =VERSION'
         DC    0D'0'              Filler
DBGLEN   EQU   *-RMTOPTS
         SPACE
         END
```

C Execution Framework Access

Once L$UMAIN has created the C execution framework, it is necessary to ensure that it is accessible when any C function executes. For a function compiled with **indep**, L$UPREP is responsible for making the C framework accessible by loading the CRAB address into register 12. Functions compiled with **indep** can thus be invoked from outside the C framework (for example, from another high-level language).

Functions compiled with **indep** can also be invoked from other C functions. In this case, the C framework is already accessible. L$UPREP is responsible for determining whether a function compiled with the **indep** option was called from C or from a non-C routine and for avoiding unnecessary or incorrect processing if the call is from C. If you modify the supplied L$UPREP routine only by replacing the L$UCENV macro, this check is performed automatically, and the code to find the C framework generated by L$UCENV is executed only for a call from non-C code.

C functions compiled without **indep** also can be used in a multilanguage environment. Because such functions do not invoke L$UPREP, they can be called only from other C functions. Calls to these functions execute slightly faster because L$UPREP is not invoked, so you may want to compile only functions that are called from C without using the **indep** option.

C Execution Framework Termination

After all C functions have completed execution and before the current task or command returns to the operating system, the C framework should be terminated. This enables memory to be freed, output buffers to be flushed, files to be closed, operating system exits such as ABNEXITs and ESTAEs to be cleared, and so on. When a C program that was compiled without **indep** terminates, the C framework is terminated automatically. When you use **indep** for calling C from another language, it usually is your responsibility to terminate the C framework after all calls to C functions have completed.

Unless the initial C function is named **main**, the C framework created by the initial call to C exists indefinitely. (See “main Function Considerations” on page 400 for information on this special case.) Therefore, subsequent C functions can access external variables set by previous functions, read and write opened files, and so on. In this case, you should ensure that the C execution framework is terminated when all C functions have completed.

To terminate the C execution framework, call the termination routine L$UEXIT. Be sure to call L$UEXIT only after all non-C routines called from C have returned; premature termination of the C framework causes errors on return from such routines. Failure to call L$UEXIT can cause incomplete file output (if files have not been closed) or can waste system resources or both. Note that you can avoid the problem of incomplete file output by closing all files yourself, including the standard files.

The execution framework may also be terminated by calling the standard **exit** function, but **exit** can be called only by a C routine, while L$UEXIT can be called by either C or non-C code.

Note that both L$UEXIT and **exit** return to the program that called the first currently active C function. Since L$UEXIT is a C function, L$UEXIT returns to the routine that called it, assuming no other C routine was active at the time of the call. Because **exit** must be called by a C function, **exit** never returns to its caller.

The name L$UEXIT cannot be referenced in FORTRAN and COBOL because of restrictions in these languages. For the convenience of users of these languages, the name CEXIT can be used instead of L$UEXIT.

Interlanguage Communication Considerations

This section discusses some programming conventions you need to consider when using interlanguage communication.

Initialization Considerations

In a sophisticated application, you may need to do additional processing after the C execution framework has been created. For example, you may need to initialize values in one or more of the user words in the CRAB. If you do this, you need to know details of the initialization process implemented by L$UMAIN.

After L$UMAIN has created the C execution framework, it calls the initial C function (the function that invoked L$UPREP) again. That function in turn passes control to L$UPREP again. Normally, at this point L$UPREP determines that the C framework is already accessible and allows the initial function to begin execution. However, you can modify L$UPREP to detect this second call and to complete its own initialization at that time. (Inspection of the sample L$UPREP source code is recommended.)

To recognize this second call and distinguish it from other calls, compare the address of the caller's DSA (R13 on entry to L$UPREP) with the contents of CRABMDSA. If they are equal, L$UPREP was invoked from L$UMAIN. This is the most reliable test because CRABMDSA usage is not expected to change in future releases of the compiler and library.

Using the longjmp Function in a Multilanguage Program

When you have a multilanguage program compiled with the **`indep`** option, you ordinarily cannot use **`longjmp`** to transfer control from one routine to another if this would terminate a non-C routine. If you attempt this, a user ABEND 1224 results. This restriction is imposed because **`longjmp`** cannot guarantee that bypassing the normal return from the non-C routine will not cause random failures in non-C code.

If your application requires the use of **`longjmp`** in this fashion, you can modify the L$UPREP routine to remove this restriction. However, if you do so, be aware that this may cause unpredictable results later in execution, depending on the implementation of function linkage in the other language.

Reentrancy

L$UPREP uses an assembler macro named L$UCENV to locate the area where the CRAB address is saved. The L$UCENV macro sets R12 to the address where the CRAB address should be saved. As provided, the L$UCENV macro uses a CSECT named L$UCENV to store the CRAB address. For this reason, programs that use L$UPREP as provided are self-modifying and, therefore, are non-reentrant.

If your application requires reentrancy, you need to modify L$UPREP to save the CRAB address in a different manner. For example, if the language that calls C provides a user word, as PL/I does, L$UPREP can be made reentrant by modifying it to save the CRAB address in the user word. In most cases, a modification to L$UCENV is all that is required to achieve reentrancy.

Note that the version of L$UPREP provided by CICS stores the CRAB address in the first word of the Transaction Work Area (TWA) so that CICS application programs compiled when the **`indep`** option is specified will be reentrant; this is a CICS requirement. A transaction which invokes a C program compiled with the **`indep`** option must be defined with a TWA.

Note: Programs that modify external data such as FORTRAN COMMONs or PL/I STATIC EXTERNAL variables are inherently non-reentrant. There is no point in making L$UCENV reentrant in such programs. △

main Function Considerations

Ordinarily, when you call C functions from another language, there is no C **main** routine because the program is composed of a main routine in the other language combined with C and other subroutines. Sometimes it is necessary to have a C **main** function, usually because an existing C program is being modified to be called from another language. This section describes the special considerations in such a case.

When you link code compiled with **indep** with a C function named **main**, the **main** function normally must also be compiled with **indep**. This means that L$UPREP is called to create the C execution framework if it does not exist and to save the CRAB address for later access by the L$UCENV macro. In this special case, the C execution framework is terminated when **main** returns. Therefore, you do not need to call L$UEXIT after a call to a **main** function.

When the compiler processes a function named **main**, the corresponding external symbol is named @MAIN. When you call a **main** function compiled with **indep** from another language, you must use this entry point name. Use of the name MAIN (without @) passes control directly to the run-time library on program entry, bypassing L$UPREP and L$UCENV for framework initialization. This means that necessary information is not stored for use by later non-C calls to C.

One difficulty that can arise when you call a **main** function from another language is that storage referenced by L$UCENV may not always exist at the time that **main** is called. This situation can occur, for example, if L$UCENV references a user word provided by another language and C code is invoked before any code in the other language. In this case, if possible, you should change the processing order so that required storage can be allocated before **main** is called (by calling the other language before C).

If it is not possible to change the processing order, an alternative technique can be used in which you compile the **main** function without **indep**. Then, the C execution framework is created by L$CMAIN when the C **main** function is invoked. The fact that storage may not be available does not matter because a **main** routine that is not compiled with the **indep** option does not invoke L$UPREP. In addition, you must store the CRAB address yourself before any C function compiled with **indep** is called so that it can be accessed by L$UCENV. To do this, you need to write an assembler routine and invoke it after the other storage is accessible (for example, after the other language's execution framework has been created) and before any C functions compiled with **indep** are called.

Note: The technique above is not directly applicable to SPE. △

Using Interlanguage Communication

This section explains how to create programs that use interlanguage calls in more than one direction as well as how to share data between languages.

Calls to C

C code compiled with **indep** can be called from FORTRAN, COBOL, PL/I, or any other language that uses standard IBM 370 linkage conventions, provided that the

differences between C parameter passing conventions and those of other languages are understood. Briefly, most languages use call by reference, while C uses call by value. For example, if a FORTRAN routine calls a C program, passing an INTEGER value, the corresponding C parameter must be declared to have type **int *** (pointer to **int**), not **int**.

Calls from C

In the simple case where C is called from another high-level language and no interlanguage calls are made by C, the use of **indep** is often the only special requirement. Sometimes, however, there is a need to support calls in both directions. If direct calls from C to the other language cause no problems, you do not need to read the rest of this section.

For many languages, direct calls from C to subroutines in the other language fail to work because the called language requires access to its own execution framework. For example, PL/I code requires that register 12 address the PL/I TCA rather than the CRAB. For such languages, before calling the subroutine, you must arrange for the creation of an execution framework for the other high-level language. When the subroutine is invoked, you must arrange for the other language's framework to be in effect. When you are finished with subroutines in the other language, you must arrange to terminate the other language framework. You may need to write assembler stubs to perform some or all of these functions.

The exact details depend on the implementation of the other language and its execution framework and on any restrictions it imposes. You should read the section or sections on interlanguage communication in the appropriate manual for the other language. C is like assembler for many purposes, so you should also study the discussion on communication with assembler. The steps required are outlined here.

First, create an execution framework for the other language. The best way to do this is to call a MAIN routine written in the other language, unless the language provides a specific call to accomplish this function. The other language's framework can be created before or after the C framework. If you create it after the C framework, calling a C function compiled with **indep** returns to the C execution framework if provision has been made for this as described under "main Function Considerations" on page 400.

C functions compiled with **indep** can be called from the other language exactly as described under "Calls to C" on page 400.

You cannot call other language subroutines directly from C functions, unless the other language implementation has a facility similar to **indep**. Instead, you must call an assembler routine to switch execution frameworks. After saving the registers from C, the stub must reactivate the framework for the other language and call the appropriate subroutine. To assist you with this, a sample macro named L$UPENV is provided. This macro is in some ways the inverse of the L$UCENV macro: it determines whether the C framework is active and, if so, switches to the framework from which C was called. L$UPENV has a single argument, REG=, that specifies the register that is to contain the framework pointer. This register is restored from the save area of the routine that called the first active C routine. If this procedure is not adequate for the particular language in use, you can modify L$UPENV as necessary.

Your assembler routine should invoke the L$UPENV macro (modified as necessary) and then invoke the other language routine. Example Code 21.3 on page 403 may be helpful. You may want to write one stub per subroutine or have one stub handle multiple subroutines, as in Example Code 21.1 on page 396.

You may need to pass on parameters from the C caller to the other language subroutine. The @ operator and the **__ref** function modifier (language extensions) can be helpful in passing parameters to another language. These extensions are described in Chapter 2, "Source Code Conventions," on page 9.

The other language subroutines can call other subroutines in the other language, C functions compiled with **indep**, and assembler subroutines.

When you are through calling all other language subroutines, you need to terminate the execution framework of the other language. You should terminate the frameworks in the opposite order in which they were created. (That is, the framework created last should be terminated first.) If you terminate C first, simply call L$UEXIT from the other language as described above. To terminate the other language's framework first, return from the other language's MAIN routine. When you do this, control is returned to the point in C from which you originally invoked the MAIN routine in the other language.

Data Sharing

In general, data belonging to another language that are to be referenced from C must be accessed via pointers. FORTRAN and PL/I provide exceptions to this. If your C program is compiled with the **norent** compiler option, it can access FORTRAN COMMONs (except for dynamic COMMONs) and PL/I STATIC EXTERNAL variables as **extern** data. For details, see "Sharing External Variables with FORTRAN Programs" on page 60.

Sample Interlanguage Calls

The following paragraphs discuss an example of calling C **indep** code from PL/I. The C code in turn calls PL/I. Example Code 21.1 on page 396 manipulates bits in PL/I save areas to prevent inspection of C save areas for ON-units. Although this manipulation keeps PL/I from getting confused in most cases, it cannot be guaranteed to work in all cases or with all versions of the PL/I library.

First, the L$UCENV macro is modified to locate the user word of the PL/I TCA (TUSR). The address of TUSR is returned in register 12, which, after completion of L$UCENV, is expected to contain the address of the word in which L$UPREP should store the CRAB address. This modified version of L$UCENV also performs a secondary function, namely, storing the address of the most recent PL/I save area in a CRAB user word so it can be used later when reentering the PL/I execution framework.

Example Code 21.2 on page 402 shows this version of L$UCENV.

Example Code A6.2 L$UCENV Macro for PL/I to C

```
         MACRO
&L       L$UCENV
&L       LA    R12,X'11C'(,R12)   Access TCA user word.
         CLC   0(4,R12),=F'0'     CRAB address known yet?
         BE    CENV&SYSNDX        If NO, no place
                                    to save SAVE area
         ST    R12,8(,R13)        Save R12 for a moment.
         L     R12,0(,R12)        Find the CRAB.
         USING CRAB,R12
         ST    R13,CRABUSR1       Save SAVE area addr in CRAB.
         DROP  R12
         L     R12,8(,R13)        Restore CRAB pointer address.
CENV&SYSNDX DS 0H
         MEND
```

Second, the L$UPENV macro must be provided. The example L$UPENV works fine in this context since L$UPENV REG=12 retrieves the PL/I TCA address.

Finally, an assembler routine must be written to allow calls from C to PL/I. A sample routine to perform this service, PLISUB, is shown in Example Code 21.3 on page 403. The first argument to the procedure is the address of the PL/I routine; the remaining arguments are those to be passed to PL/I. The PLISUB routine must

1. locate the PL/I routine's actual entry point
2. switch to the PL/I execution framework by using L$UPENV
3. build a save area for itself that looks enough like a PL/I DSA to avoid confusing the PL/I library.

When the PL/I routine returns, PLISUB must put everything back before returning. A sample call to PLISUB to perform the equivalent of the PL/I call CALL TRANS(I, 2); follows:

```
extern trans();
plisub(&trans, @i, @2);
```

Example Code A6.3 Sample PLISUB Routine

```
PLISUB@   CSECT
          CREGS
          USING CRAB,R12
PLISUB    CENTRY DSA=DSALEN
          L     R2,0(,R1)     Address PL/I "function pointer."
          L     R2,0(,R2)     Address PL/I entry point.
          LA    R3,4(,R1)     Address PL/I program's parms.
          LR    R4,R12        Save CRAB address.
          DROP  R12
          USING CRAB,R4
          L$UPENV REG=12      Find the PL/I TCA.
          LR    R6,R13        Save our DSA address.
          USING DSA,R6        Establish DSA addressability.
          MVC   0(2,R6),=X'8200'   Mark it as a dummy PL/I DSA.
          LA    R13,PLIDSA    Find a save area for PL/I to use.
          XC    0(96,R13),0(R13)
          ST    R6,4(,R13)    Link to our SAVE area.
          MVC   0(4,R13),=X'80000000' INIT SAVE area for PL/I
          MVC   86(2,R13),=X'91C0'    as described in PL/I doc
          ICM   R7,15,CRABUSR1        Find last PL/I SAVE area
          BNZ   OK
          L     R7,CRABPENV   or save area on entry to C.
OK        DS    0H
          MVC   72(8,R13),72(R7) Copy PL/I storage management  O
          LR    R15,R2        Set up regs for call.
          LR    R1,R3         Put address of parms where
                                PL/I expects.
          BALR  R14,R15       Call PL/I.
          LR    R13,R6        Restore our own SAVE area
          LR    R12,R4        and R12.
          ST    R7,CRABUSR1   Restore CRAB PL/I SAVE area ptr.
          CEXIT ,             Return to C.
          SPACE
```

```
          LTORG
          COPY  CRAB
          SPACE
          COPY  DSA
          SPACE
PLIDSA    DS    CL96          Space for PL/I SAVE area.
          SPACE
DSALEN    EQU   *-DSA
          END
```

Link-Editing Multilanguage Programs

If your program is to be executed using the normal C framework, use the normal C resident library files as your autocall libraries (SASC.BASEOBJ and SASC.STDOBJ under OS/390, LC370BAS TXTLIB and LC370STD TXTLIB under CMS). These libraries include versions of L$UMAIN, L$UEXIT, and so on, suitable for use with the full execution framework. For example, this version of L$UMAIN creates a complete C execution framework, allowing the full use of all library functions.

Do not use the SPE object library when link-editing a program that is to execute with the normal C framework because you then obtain versions of L$UMAIN and L$UEXIT that are incompatible with the normal C framework.

Note that when you use the normal C framework, L$UPREP is the only interlanguage communication routine that can be modified. If you have modified L$UPREP, you can either replace the copy in the base library, as described earlier, or you can store the modified version in a separate library and include that library in the GLOBAL TXTLIB list (CMS) or SYSLIB concatenation (OS/390) before any other autocall libraries. (The latter is recommended for safety.) The sample L$UPREP can be used with either the full framework or with SPE. If you modify L$UPREP so that it depends on one of these two frameworks, be careful to always link with the version that is appropriate for your application.

If you write an L$URTOP routine, its object code also can be stored in the base resident library. However, usually this routine is highly application-specific, so it is probably more appropriate to store it in a separate library.

Location of indep Libraries

The sample source code for L$UPREP can be found in LSU MACLIB (CMS) or SASC.SOURCE (OS/390). The object code is in LC370SPE TXTLIB (CMS) or SASC.SPEOBJ (OS/390). The macro library is LCUSER MACLIB (CMS) and SASC.MACLIBA (OS/390). The library contains macros such as L$UCENV and L$UPENV. The macro library also contains members CRAB, DSA, and CPROLOG, which provide mappings of the minimal CRAB, the DSA (Dynamic Save Area), and the compiler-generated prolog code, respectively, which may be helpful in understanding the sample routines or in coding new ones.

APPENDIX

7

Extended Names

Introduction

This appendix explains how the compiler and the COOL utility work together to allow extended names to be used in C programs. An extended name is a C identifier with external linkage that is either more than eight characters in length or that contains uppercase characters.

Extended Names Processing

The **extname** compiler option specifies that the compiler encode extended names in eight-character, uppercase external symbols and save the original extended names in generated object code. When the COOL utility processes object modules that contain extended names, it uses the saved extended names to create external symbols that allow the linker to link object files correctly. If **extname** is not specified, the compiler creates external symbols for extended names by converting and truncating each name to a maximum of eight uppercase characters.

The **extname** option directs the compiler to allow all identifiers, whether internal or external, to be up to 65,535 characters in length. The compiler refers to internal identifiers by their full names. During object code generation, the compiler examines the identifier of each **extern** variable and each **extern** or **static** function and determines if the identifier meets one or both of the following criteria:

- It is more than eight characters in length.
- It contains at least one uppercase alphabetic character. This criterion ensures that the compiler will distinguish between identifiers that differ only in case, for example, between the function names **gets** and **Gets**.

A name is an extended name if either of these conditions is true, and it is not the name of a function declared with the **__asm**, **__ref**, or **__ibmos** keyword or one of the high-level language keywords, such as **__pascal**.

Note that all function names, either **static** or **extern**, may be considered extended names. Even though the name of a **static** function is not visible externally, the extended function name is retained for use in commands to the debugger.

If the name is an extended name, the compiler assigns to it an eight-character external symbol that represents it in object code. A unique extended name is always assigned the same symbol, no matter how often it occurs within a compilation. The external symbol is in the following form:

```
@@nnnnnn
```

where *nnnnnn* is a decimal number. The COOL utility uses this number to find the extended name associated with the external symbol.

When the **extname** option is in effect, the compiler is sensitive to case with regard to the special function names **main** and **_dynamn**. In other words, a program can have both a **main** and a **MAIN** function. The compiler will treat only the **main** function as the main entry point. When the **noextname** option is specified, the compiler is case insensitive with regard to the special function names **main** and **_dynamn**. That is, the compile r will accept **MAIN** as the main entry point and **_DYNAMN** as the dynamically loaded entry point.

When the **norent** compiler option is specified, the compiler may also create external symbols in the form of **&@nnnnnn**. These symbols represent a pointer to a function with an extended name.

Extended Names CSECTs

The compiler stores each extended name in one of two extended names CSECTs in the output object module. One of the extended names CSECTs contains the extended names of all of the **static** and **extern** functions defined in the compilation. The other extended names CSECT contains the extended names of all other external identifiers. The names of both CSECTs are formed from the compilation section name. The suffix character for the extended function name CSECT is a right angle bracket (>). The suffix character for the extended external identifier CSECT is a left angle bracket (<). Refer to “Control Section Names” on page 53 for more information on CSECT names and how they are formed.

CSECT Format

Both extended names CSECTs have the same format. A representation of the extended names CSECT format in assembler language follows:

```
SNAME@>  CSECT
         DC    F'nnnn'
```

```
         DC   XL2'length-1 ',C'name-1'
         DC   XL2'length-2 ',C'name-2'
         .
         .
         .
         DC   XL2'0'
         END
```

The CSECT begins with a fullword value. Following this value is the length of the first extended name, *length-1*, represented as an unsigned halfword. The extended name, *name-1*, follows the length. The next extended name, *name-2*, follows the same format. The extended names are not terminated by nulls. Following all of the extended names is a halfword with all bits set to 0.

Extended external identifiers CSECT

The fullword field at the beginning of the extended external identifiers CSECT contains the minimum external symbol value, that is, the smallest value of external symbol nnnnnn used to create the external symbols for the extended external identifiers. The compiler assigns external symbols beginning with 750000 and increments by 1 for each extended external identifier. For example, the compiler assigns @@750000 to the first extended external identifier, @@750001 to the second, and @@750002 to the third. Because the maximum value for nnnnnn is 999999, there can be no more than 250,000 extended external identifiers, excluding function names, in a single compilation. This is also the maximum number of extended external identifiers in a load module.

Extended function names CSECT

In the extended function names CSECT, the fullword field contains a number created by hashing the compilation section name. The hash value is in the range from 0 through 749999. For a given external function name, the compiler determines an identifying number n for the function by adding the hash value to the offset of the length field of the external name in the CSECT. If n is greater than 749999, the compiler uses the offset in the CSECT as n, ignoring the hash value. In either case, the external name assigned by the compiler is @@NNNNNN, where NNNNNN is the decimal expansion of n, padded on the left with zeroes if necessary.

The following example illustrates the process of deriving external symbols for extended function names:

```
SNAME@>  CSECT
         DC   F'2456'
         DC   XL2'22',C'My_Structure_Type_Copy'
         DC   XL2'26',C'My_Structure_Type_Allocate'
         DC   XL2'24',C'My_Structure_Type_Delete'
         DC   XL2'0'
         END
```

The external symbol for *My_Structure_Type_Copy* is the offset of the length field in the CSECT, 2456+4, or @@002460. The external symbol for *My_Structure_Type_Allocate* is 2456+26, or @@002482. The external symbol for *My_Structure_Type_Delete* is 2456+52, or @@002508. The total number of extended function names in a compilation depends on the cumulative length of the function names; no extended function name can begin at a location greater than 750000 bytes from the start of the CSECT. However, this is not the limit of extended function identifiers in a load module.

The compiler also stows the external symbol in the function name field of the function prolog. This copy of the external symbol can be used at runtime to associate a function entry point with the original extended function name.

The enxref Compiler Option

When the **enxref** compiler option is specified, the compiler creates two extended name cross-references. Both of these cross-references are in two-column format. The first cross-reference is sorted according to extended name, or C identifier. The second cross-reference is sorted according to external symbol, or Link ID. Both cross-references include only the extended names of objects that were defined in the compilation; a **__rent** external variable is considered to be defined even if its only usage in the compilation is as a reference. Both of these cross-references are of limited value because the COOL utility can assign different external symbols to extended names during object code processing. The cross-references produced by COOL are more meaningful.

COOL Extended Names Processing

The external symbol that the compiler assigns for an extended name is different for each compilation in which the name appears. For example, the external symbol for the definition of the **my_structure_allocate** function is based on the sum of the section name hash value and the offset of the extended name in the extended function names CSECT. The external symbol for a call to the same function in another compilation uses a number between 750000 and 999999.

The COOL object code preprocessor collects all the extended names in all the object files for a load module and assigns a unique external symbol to each extended name. The COOL utility preprocesses all object files that contain extended names CSECTs.

When the **extname** option has been specified and COOL detects an extended names CSECT in an input object file, COOL creates a table of external symbols and associated external names. COOL selects a unique external symbol for each extended name after all of the input files have been processed and then uses this external symbol to replace the compiler-assigned external symbol in the object code. Extended names processing is the default. Specify **noextname** to suppress extended names processing.

This paragraph gives an example of the way COOL assigns external symbols. In the object file for compilation A, the compiler uses the external symbol @@750078 to refer to **Instance_Number**. In the object file for compilation B, the compiler uses @@750012 to refer to the same name. While processing the two object files, COOL selects a third external symbol, @@750341, to represent **Instance_Number** and replaces all instances of @@750078 in compilation A and all instances of @@750012 in compilation B with this selection. The linkage editor can then link the load module correctly because the preprocessed object code contains a unique eight-character monocase external symbol for each extended name.

Note: If the ability to autocall a function with an extended name is a requirement and you cannot store the object code in an AR370 archive, you can use the **#pragma map** directive to map the symbol to an eight-character name. (See Appendix 2, "The AR370 Archive Utility," on page 353.) △

COOL Selection of External Symbols

COOL selects external symbols differently, depending on the type of extended name. For all extended names other than function names, COOL assigns external symbols by starting with @@750000 and incrementing by 1 for each new name. COOL changes all of the external symbols for all extended names, other than function names, that were assigned during compilation. There can be no more than 250,000 extended external identifiers in a load module.

For extended function names, COOL attempts to use the external symbol that the compiler assigns to the function definition. Because this symbol has been stored in the function prolog, all references to the function can use the same symbol. In the unlikely event that the compiler has assigned the same external symbol to more than one function, COOL selects the next higher unassigned value as the external symbol.

This paragraph gives an example of the way in which COOL assigns external symbols in this instance. The compiler has assigned @@189676 as the external symbol for both **function_A** in compilation A and **funct ion_B** in compilation B. COOL assigns a new external symbol for the second of these two functions that it finds in the object code. If it has not already been used, COOL assigns the next higher symbol, @@189677, to the second function.

User Exit Selection of External Symbols

COOL can call a user exit to create the external symbols instead of creating them itself. The exit supplies the numerical part of the external symbol in the form of an integer. The exit may choose the numbers using any algorithm, although it is assumed that in most cases it will reference a database of extended names. This functionality may be useful if the same external symbol is to be assigned to the same extended name in multiple load modules.

ENEXIT and ENEXITDATA options

The exit is enabled by one of two options. The first option is the **enexit** option. This option causes COOL to invoke the exit. The second is the **enexitdata** option. This option causes COOL to invoke the exit and pass one to eight characters of user-specified data. In TSO and OS/390 batch, this option has the following form:

```
enexitdata(userdata)
```

Under UNIX System Services (USS), the option is specified as follows:

```
-Aenexitdata=userdata
```

Under CMS, the option is specified as follows:

```
enexitdata userdata
```

The data are intended to give the exit a way to identify the object files that COOL is processing. For example, the *userdata* could be a load module name.

If either option is used, COOL tries to dynamically load a module named CLKEXIT. In TSO or OS/390 batch, this module must be located in STEPLIB, a task library, or the system link list. Under CMS, the module must be a member of a LOADLIB named DYNAMC LOADLIB. If the load module cannot be loaded, COOL terminates with an error message.

The exit is called once for each unique extended name in the COOL input files. The exit must return a valid value for every extended name. It is not possible for the exit to elect to return values for a proper subset of the extended names.

The exit itself takes the form of a dynamically loadable function having the prototype shown here:

```
int _dynamn(char UserData[8],
            const char *ExtendedName, int ExtendedNameLength,
            int FunctionFlag, int OldId, unsigned *NewId);
```

UserData is a pointer to an eight-byte array containing the one to eight characters of user data specified by the **enexitdata** option. Unused characters in the array are set to blanks. The exit can change the values in the **UserData** array. COOL always passes

a pointer to the same array, so any changes made during a call to the exit are carried over to the next call. **`ExtendedName`** is a pointer to the extended (long) name. **`ExtendedNameLength`** is the length (1 to 65,535) of the extended name. **`FunctionFlag`** is set to a non-zero value if the name is an extended function name and is 0 otherwise. If **`FunctionFlag`** is nonzero, then **`OldId`** contains the integer part of the compiler-assigned external symbol. If **`FunctionFlag`** is 0, then the value in **`OldId`** has no meaning. **`NewId`** is a pointer to an **`unsigned int`**, into which the exit should store the integer part of the external symbol to be assigned by COOL. The value stored in **`NewId`** must be between 0 and 999999 inclusive. If the exit returns a value that is not in this range, COOL terminates. The return code values are as follows:

0	indicates normal return. The exit supplied a value for **`NewId`**.
4	if returned on the first call to the exit, indicates that CLINK should continue normal extended names processing and not call the exit again.

Any other return code causes COOL to terminate immediately.

The exit runs as a function called from COOL; therefore, if the exit cannot continue execution, it can call the **`exit`** or **`abort`** functions. The exit can issue diagnostic messages by writing to STDERR. The exit should not write to STDOUT or any other file that may write to the DDname SYSPRINT because this may interfere with COOL's use of this DDname.

The enxref COOL Option

COOL can optionally produce three extended names cross-references: SNAME, CID, and LINKID. The **`enxref`** option controls these three cross-references. The SNAME cross-reference is the most informative.

The SNAME Cross-Reference

The extended names in the SNAME cross-reference are sorted by compilation SNAME. The extended names are in alphabetical order within each SNAME. The SNAME cross-reference is displayed in two columns. Example Code 22.1 on page 410 shows one column of a sample SNAME cross-reference.

Example Code A7.1 Sample SNAME Cross-Reference

```
C IDENTIFIER       (SNAME=PROGRAM)        LINK ID     WAS
Get_Option_String.   .   .   .   .   .    @@002512    (same)
Initialize_Printer   .   .   .   .   .    (static)   @@002488
Instance_Number  .   .   .   .   .   .    @@750321   @@750017
My_Structure_Type_Allocate   .   .   .    @@002460   @@750012
```

The three fields in this example show, from left to right, the original extended name in the C IDENTIFIER field, the external symbol assigned by COOL in the LINK ID field, and the external symbol assigned by the compiler in the WAS field. If the extended name is the name of a **`static`** function, that is, a function that is not called from any compilation other than the one in which it is defined, **`(static)`** is placed in the LINK ID field. If COOL used the external symbol assigned by the compiler for an extended function name, **`(same)`** is placed in the WAS field.

The SNAME cross-reference includes only those extended names that are defined in the compilation. As in the extended name cross-references for the compiler, a reentrant external variable is considered to be defined in every compilation that references it.

The CID Cross-Reference

The second cross-reference is the CID cross-reference. In this cross-reference, the extended names are shown in alphabetical order by original extended name. The CID cross-reference is also displayed in two columns. Example Code 22.2 on page 411 shows one column of a sample CID cross-reference.

Example Code A7.2 Sample CID Cross-Reference

```
C IDENTIFIER                          LINK ID
Get_Option_String.   .   .   .   .    @@002512
Instance_Number  .   .   .   .   .    @@750321
My_Structure_Type_Allocate   .   .    @@002460
```

Each column of the CID cross-reference contains two fields. The C IDENTIFIER field contains the original extended name. The LINK ID field contains the external symbol assigned by COOL. The names of **`static`** functions are not listed in the CID cross-reference.

The LINKID Cross-Reference

The third cross-reference is the LINKID cross-reference. It is also displayed in two columns. The entries are sorted by external symbols that COOL assigns. Example Code 22.3 on page 411 shows one column of a sample LINKID cross-reference.

Example Code A7.3 Sample LINKID Cross-Reference

```
LINK ID                               C IDENTIFIER
@@002460 .   .   .   .   .   .   .    My_Structure_Type_Allocate
@@002512 .   .   .   .   .   .   .    Get_Option_String
@@750321 .   .   .   .   .   .   .    Instance_Number
```

Each column of the LINKID cross-reference contains two fields. The LINK ID field contains the external symbol assigned by COOL. The C IDENTIFIER field contains the original extended name. The names of **`static`** functions are not listed in the LINKID cross-reference. See for more information on C IDENTIFIER lengths.

The xfnmkeep Option

The **`xfnmkeep`** option retains the extended function names CSECTs in all input object files. The extended function names CSECT may be useful at runtime if you are using the SAS/C Debugger to debug your program. If the CSECT containing the extended function name is available, the debugger uses the extended name in displays and accepts the extended name in commands. Refer to the SAS/C Debugger User's Guide and Reference for more information on the debugger. Also, if the CSECT that contains the extended function name is present, the library ABEND-handler includes the extended name in abend tracebacks.

The **xfnmkeep** option is the default. If **noxfnmkeep** is specified, COOL deletes the extended function names CSECTs after the input files are processed. These CSECTs will not appear in the output object file.

The xsymkeep Option

The **xsymkeep** option specifies that the extended external identifier CSECTs in all input files are retained. The default is **noxsymkeep**. The **noxsymkeep** option specifies that the extended external identifier CSECTs are not retained in the output object module.

Note that retaining the extended function names CSECT or the extended external identifier CSECT makes the resulting load module somewhat larger.

Determining the Extended Function Name at Execution Time

You can determine the extended name of a function at execution time by taking the following steps. Note that all offsets are in decimal.

1. Find the external symbol name, stored at offset 5 in the function prolog (except in a very large function). During execution, general register 5 points to the start of the function prolog. Alternately, you can access the function entry point via the register 15 value stored at offset 16 in the previous save area. This technique is effective regardless of function size.
2. Get the address of the constants CSECT from offset 32 in the function prolog or from general register 4.
3. Determine the address of the run-time constants CSECT. This address is located at offset 8 in the constants CSECT.
4. Determine the address of the extended function names CSECT. This address is located at offset 12 in the run-time constants CSECT.
5. Retrieve the fullword value (the SNAME hash value) located at offset 0 in the extended function names CSECT.
6. Convert the six digits of the external symbol to binary.
7. If the result of this conversion is greater than the SNAME hash value, subtract the hash value. The result is the offset of the extended name in the CSECT. The first halfword at this offset is the length of the extended name followed by the extended name itself.

To get a clearer picture of the content of these CSECTs, examine an OMD370 listing. Use the **verbose** option to include the extended names CSECTs in the disassembly. For more information about the constants CSECT and the run-time constants CSECT, refer to “Compiler-generated Names” on page 53. Chapter 3, “Code Generation Conventions,” on page 45 also contains more information about CSECT addressing at execution time.

The PRTNAME Function

The PRTNAME function is an assembler language function that prints the name of the function pointed to by its argument. This function illustrates the process explained above. The prototype for PRTNAME is the following:

```
void prtname(_remote void (*) (void));
```

If the argument points to a function with an extended name, the extended name is printed. Otherwise, the name stored in the function prolog is printed.

Example Code 22.4 on page 413 shows the PRTNAME function.

Example Code A7.4 The PRTNAME Function

```
PRTNAME@ CSECT
         CREGS USING
         USING CPROLOG,R2
PRTNAME  CENTRY DSA=DSALEN
         L     R2,0(,R1)    R2 -> function pointer
         L     R2,0(,R2)    R2 -> function prolog
         LA    R3,CPROFNM   R3 -> function name
         CLC   =C'@@',CPROFNM  Start with @@?
         BNE   NOTEXTND     No - not an extended name
         L     R4,CPROCONS  R4 -> constant CSECT
         L     R4,8(,R4)    R4 -> run-time constants CSECT
         L     R4,12(,R4)   R4 -> extended function names CSECT
         LTR   R4,R4        Set to 0 if CSECT not present.
         BZ    NOTEXTND
         L     R0,0(,R4)    R0 = SNAME hash value
         PACK  EXTSYM(8),CPROFNM+2(6)  Convert 6 digits in external
         CVB   R5,EXTSYM     symbol name to binary.
         CR    R5,R0        Is symbol less than hash value?
         BNH   FOUND        If so, don't subtract.
         SR    R5,R0        If not, subtract hash value.
FOUND    DS    0H
         AR    R5,R4        R5 -> length of extended name
         LH    R0,0(,R5)    R0 = length
         ST    R0,NAMELEN   Store in printf parm list.
         LA    R5,2(,R5)    R5 -> extended name
         ST    R5,NAME      Store in printf parm list.
         B     CALLPRTF     Go call printf.
*
* Handle function names that aren't extended.
*
NOTEXTND DS    0H
         MVC   NAMELEN,=F'8' Store length in plist.
         ST    R3,NAME      Store pointer to name in plist.
*
* Call printf to print the function name.
*
CALLPRTF DS    0H
         MVC   FORMAT,=A(FMTSTR)
         LA    R1,PARMLIST
         L     R15,=V(PRINTF)
         BALR  R14,R15
         CEXIT DSA=DSALEN,RC=0
         DROP  R2
         LTORG
FMTSTR   DC   C'"%.*s"',X'1500' printf format
         COPY DSA
EXTSYM   DS   8C            decimal-to-binary conversion area
PARMLIST DS   0F            printf parameter list
```

```
FORMAT   DS   A
NAMELEN  DS   A
NAME     DS   A
DSALEN   EQU  *-DSA
         COPY CPROLOG
         COPY CRAB
```

Two tests are performed to discover if the function has an extended name. The first test determines if the name stored in the function prolog begins with **@@**. The second test determines if the address of the extended function names CSECT is nonzero. If the **NOXFNMKEEP** COOL option is specified, this address is set to 0, and the extended function names CSECT is deleted from the output object file.

Using #pragma map to Create Constant External Symbols

It is impossible to predict what external symbol the compiler or COOL will create for an extended name. This unpredictability can cause problems when a constant, predictable symbol is required for a name.

For example, the ANSI Standard function name **localtime** is nine characters in length. The **extname** compiler option would treat this name as an extended name. The program that calls **localtime** must have a predictable symbol for the name or it will not be possible to link the **localtime** function into the program properly.

The **#pragma map** statement assigns an external symbol name to an extended name. For example, the following statement assigns the external symbol name **#LOCALTM** to the **localtime** extended name:

```
#pragma map (localtime, "#LOCALTM")
```

In the object file generated by the compiler, the external symbol for **localtime** is **#LOCALTM**, not **@@xxxxxx**. For more information on **#pragma map**, refer to Chapter 2, "Source Code Conventions," on page 9.

Note: Use of the **enexit** option, described in "COOL Options" on page 149 may be a more useful technique than **#pragma map** for very large applications or for applications where the external names in use are not easily predictable. △

Extended Names Processing by the GATHER Statement

In addition to normal GATHER processing, COOL inspects extended names when it gathers external symbols under the direction of the GATHER control statement. If COOL encounters a name that matches a GATHER prefix exactly, including case, COOL gathers the external symbol for the extended name.

For example, suppose an input object file contains the extended name **Step_Into**, which has an associated external symbol **@@024561**, and the following GATHER statement is used:

```
GATHER Step
```

When COOL processes the object file, **@@024561** will be gathered because the prefix of **Step_Into** exactly matches **Step**. The external symbol associated with a function named **stepOver** would not be gathered because the function name does not begin with the exact prefix specified in the GATHER statement.

APPENDIX

8

Library Initialization and Termination Exits

Introduction

You can instruct the SAS/C Library to call a site exit routine while the C framework is being created and before the initial user function is called; the library can also call a site exit routine during destruction of the C framework after normal program completion. These exit routines can be used for accounting, to define user-signal processing, or for other special purposes. They can be written in either C or assembler language and can use any C facilities. The library uses dummy exits if you do not provide any.

Note: Initialization and termination exits are not supported in the minimal SPE framework. △

Location of Exits

Copies of L$XSTRT and L$XFINI reside in SASC.STDOBJ/STDLIB, SASC.GOSOBJ, and SASC.CICSOBJ/CICSLIB under OS/390 and in LC370STD TXTLIB, LC370GOS TXTLIB, and LC370CIC TXTLIB under CMS. To install site versions of the exits, replace the members L$XSTRT and L$XFINI with the object code for your routines. You must replace both L$XSTRT and L$XFINI if you replace either one.

Because the standard library, the GOS library, and the CICS library are independent of one another, you can replace the exits in one or more of these libraries without replacing the exits in all of them. You can also install different exits in each library to target different systems.

Exit Linkage Conventions

The following sections describe the linkage conventions for the library initialization and termination exits.

L$XSTRT

The SAS/C initialization exit, L$XSTRT, is called by library initialization shortly before control is passed to **main**. If the **indep** compiler option is specified, the library calls this exit before control is passed to the first user function.

Linkage for L$XSTRT is defined as follows:

```
void L$XSTRT(void *user_words [4] );
```

user_words is a pointer to four words that can be modified to contain information that you specify. On return from L$XSTRT, these words are copied to the four user words in the CRAB that are available to the user and the installation. The program can access these CRAB user words during execution. Refer to "The C Run-Time Anchor Block" on page 215 for more information on the CRAB.

L$XFINI

The SAS/C termination exit, L$XFINI, is called by library termination after all **atexit** routines have been called but before files are closed or signal handling is terminated. L$XFINI receives the program's exit code as a parameter and can change it.

Linkage for L$XFINI is defined as follows:

```
int L$XFINI(void *user_words [4] , int rc);
```

user_words is a pointer to four words that contain the current contents of the CRAB user area. These values are the same as those stored by L$XSTRT, unless the CRAB user words were modified during program execution. The **rc** parameter contains the program's exit code. The library takes the return value from L$XFINI library as an overriding exit code, replacing the exit code specified by the program. If you do not want the program's exit code to be changed, specify **return (rc);** within the program so that L$XFINI will return the value of **rc**. The library calls L$XFINI only once during program termination regardless of the number of times that exit is called from the program.

Note that L$XFINI is called only when execution is terminated normally, due to a call to exit or L$UEXIT or due to return from the **main** function. It is not called if the program is terminated by an ABEND or an unhandled UNIX System Services (USS) signal.

APPENDIX

9

SAS/C Redistribution Package

Introduction

To facilitate the distribution of your SAS/C applications, you may need to redistribute certain files provided by SAS Institute. The files provided by the SAS/C Limited Distribution Library are redistributable on an "as is" basis. You may also want to redistribute files that are included in the SAS/C®; Redistribution Package. Licensing the SAS/C Redistribution Package allows you to redistribute a selection of SAS/C programs and libraries to your customers, above and beyond the files provided by the SAS/C Limited Distribution Library. Available for OS/390 and CMS, the SAS/C Redistribution Package may only be licensed by current SAS/C Compiler sites.

Limited Distribution Library

The SAS/C Limited Distribution Library files are redistributable on an "as is" basis. These files are copied to tape by running one of the following batch jobs:

- Under OS/390, run the JCL contained in *sasc.cntl* (DUMPRLDB).
- Under CMS, run the DUMPRLDB EXEC.

The files copied to tape by these batch jobs contain all of the SAS/C programs and libraries that are redistributable at no charge. To redistribute other SAS/C programs and libraries you must license the SAS/C Redistribution Package.

To obtain a list of the files that are written to tape by your job, print a listing of the JCL or EXEC. On OS/390, the JCL can be found in *sasc*.CNTL(DUMPRLDB), where the *sasc* qualifier is site-specific. If you cannot locate the JCL or EXEC, please see your SAS Support Consultant or Installation Representative for site-specific information.

SAS/C Redistribution Package

This section lists the programs and libraries that comprise the SAS/C Redistribution Package. This list is subject to change at any time. For more information about

redistribution, have your SAS/C Software Consultant or Representative call the Institute's Technical Support Division. For additional information regarding the terms and conditions under which these programs and libraries may be redistributed, please refer to the SAS/C Compiler Supplement to your Master License Agreement.

OS/390 Components

The following table lists the programs that comprise the OS/390 components of the SAS/C Redistribution Package:

Table A9.1 OS/390 Components of the SAS/C Redistribution Package

File	Description
sasc.BASELIB	Base resident library, load module format
sasc.BASEOBJ	Base resident library, object module format
sasc/bin/**`binder`**	USS binder front-end
sasc/bin/**`pdscall`**	USS program call utility
sasc/bin/**`sascc370`**	USS COOL/binder front-end
sasc.CICS.SPELIB	CICS SPE resident library, load module format
sasc.CICS.SPEOBJ	CICS SPE resident library, object module format
sasc.CICSLIB	CICS resident library, load module format
sasc.CICSOBJ	CICS resident library, object module format
sasc.CLIST	CLIST for compiler, other utilities
sasc.CNTL	Installation JCL, DUMPRLDB job
sasc.EXEC	REXX EXECs for compiler, other utilities
sasc.GOSOBJ	Generalized Operating System library
sasc.HELP	TSO help files for CLISTs
sasc.ILCOBJ	Interlanguage Communication library, object module format
sasc.ILCSUB	Interlanguage Communication library, load module format
sasc.LIBCXX.A	C++ resident library
sasc.LOAC(AR370#)	AR370 utility
sasc.LOAD(ILCL)	ILCLINK utility
sasc.LOAC(CLINK)	CLINK utility
sasc.LOAD(CLK370B)	CLINK batch fron-end
sasc.LOAD(COOL#)	COOL utility
sasc.LOAD(COOLB)	COOL batch front-end
sasc.LOAD(SHELLER)	C++ template utility
sasc.PROCLIB	JCL for compiler, other utilities
sasc.SOURCE	Library source code
sasc.SPELIB	SPE resident library, load module format

File	Description
sasc.STDLIB	Standard resident library, load module format
sasc.SPEOBJ	SPE resident library, object module format
sasc.STDOBJ	Standard resident library, object module format
sasc.VSEOBJ	CICS/VSE resident library

Note: Under OS/390, the first level qualifier, sasc in Table A9.1 on page 418, is site-specific. See your SAS/C Software Consultant or Representative for the qualifier used at your site. △

CMS Components

The following table lists the programs that comprise the CMS components of the SAS/C Redistribution Package:

Table A9.2 CMS Components of the SAS/C Redistribution Package

Filename	Filetype	Description
LC370BAS	TXTLIB	Base resident library
LC370CIC	TXTLIB	CICS resident library
LC370GOS	TXTLIB	Generalized Operating System Library
LC370SPC	TXTLIB	CICS SPE resident library
LC370SPE	TXTLIB	CICS SPE resident library
LC370STD	TXTLIB	Standard resident library
LC370VSE	TXTLIB	CICS/VSE resident library
L$I*	TEXT	Interlanguage Communication TEXT files
LCXX370	A	C++ resident library
SHELLER	MODULE	C++ template utility
LSU	MACLIB	Library source code
ARLIST	EXEC	AR370 utility
ARLIST	HELPC	
$PROFAR$	XEDIT	
$PROFSL$	XEDIT	
AR370	MODULE	
ILCLINK	EXEC	ILCLINK utility (370 mode CMS)
ILCLINK	HELPLC	
ILCLINK	MODULE	
ILCLINK	EXEC	ILCLINK utility (CMS)
ILCLINK	HELPLC	
ILCLINK	MODULE	

Filename	Filetype	Description
COOL	EXEC	COOL utility
COOLS	MODULE	
	HELPLC	
COOL	LOADLIB, members	
LC370	COOL and CLINK	
CLINK	EXEC	CLINK utility
CLINK	MODULE	
CLINK	HELPLC	

Index

Special Characters

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Your Turn

If you have comments or suggestions about SAS/C Compiler and Library 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.