SAS/C® Cross-Platform Compiler and C++ Development System 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 Cross-Platform Compiler and C++ Development System User's Guide, Release 7.00, Cary, NC: SAS Institute Inc., 2001.

SAS/C Cross-Platform Compiler and C++ Development System User's Guide, Release 7.00

ISBN 1–58025–730–5

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

1st printing, April 2001

Contents

P A R T 1

Usage Guide

CHAPTER 1 Overview of the SAS/C Cross-Platform Compiler and C++ Development System

Introduction

The SAS/C Cross-Platform Compiler and the SAS/C Cross-Platform C++ Development System* runs on a workstation and produces prelinked output files that can be transferred to an IBM® System/370™ mainframe. On the IBM System/370 mainframe, the files can be linked to produce an executable load module.

Like the SAS/C Compiler, the SAS/C Cross-Platform Compiler and C++ Development System is a portable implementation of the high-level C and C++ languages. It provides the same function under the UNIX, Windows 95, and Windows NT** operating systems as the SAS/C C and C++ Development System does under the OS/390 or CMS operating systems. This includes the following:

□ All SAS/C C and C++ preprocessor capabilities.

□ All SAS/C C and C++ code generation capabilities, including optimization and other features.

□ All of the prelinking functions provided by the SAS/C COOL utility, including extended names processing.

Note: Starting with Release 6.00, COOL replaced CLINK as the default SAS/C prelinker on OS/390 and CMS systems. Similarly, the program **`cool`** replaced

* For future references, SAS/C C and C++ is synonymous with SAS/C Cross-Platform Compiler and SAS/C Cross-Platform C++ Development System.

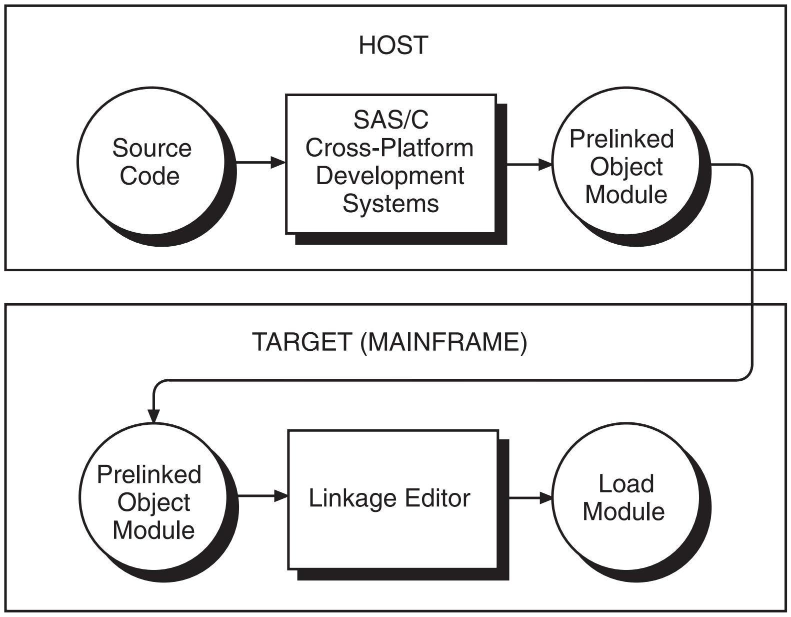
** For future references, Windows is synonymous with Windows 95 and Windows NT.

clink as the default prelinker for the SAS/C Cross-Platform Compiler and C++ Development System. For details on this and other changes in Release 6.50, see Appendix 5, "Compatibility Notes," on page 145. △

- □ An object module disassembler that can be used to generate an object module disassembly listing file.
- □ The capability of producing object modules suitable for debugging with the SAS/C Debugger.
- □ Support for the SAS/C CICS preprocessor.
- □ Support for portable **ar370** archive libraries.

Figure 1.1 on page 4 illustrates the application development process using the SAS/C Cross-Platform Compiler and C++ Development System.

Figure 1.1 Application Development Process



Why Use the SAS/C Cross-Platform Compiler and C++Development System

There are several benefits to using the SAS/C Cross-Platform Compiler and C++ Development System to generate code for the mainframe.

Reduced mainframe load
: By moving compilations off of 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 hierarchical file systems.

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

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

System Requirements

Before using the SAS/C Cross-Platform Compiler and C++ Development System, you must consider the system requirements of both the host workstation that runs the cross-platform development system and the target mainframe that runs the applications developed with the cross-platform development system.

Workstation (Host)

The host workstation provides the development platform used to run the cross-platform development system. Release 6.50 of the SAS/C Cross-Platform Compiler and C++ Development System runs under a UNIX operating system on a Sun-4, a Sun-5, an RS/6000, or under LINUX. It also runs under a DOS shell or the Microsoft Visual C++ Integrated Development Environment (IDE) on a Windows workstation running Windows 95 or Windows NT. Future releases of the cross-platform development system may support additional platforms or other operating systems.

Mainframe (Target)

The intended target for applications developed with the cross-platform development system is an IBM System/370 mainframe running either OS/390 or CMS. Your applications can be redistributed in accordance with the restrictions described in Appendix 4, "Redistributing SAS/C Product Files," on page 135.

File Transfer

In addition to the host and target system requirements, you must also consider the file transfer system that will be used to move your prelinked output files from the host to the target. One of the most effective methods of transferring files between systems is file transfer protocol (FTP).

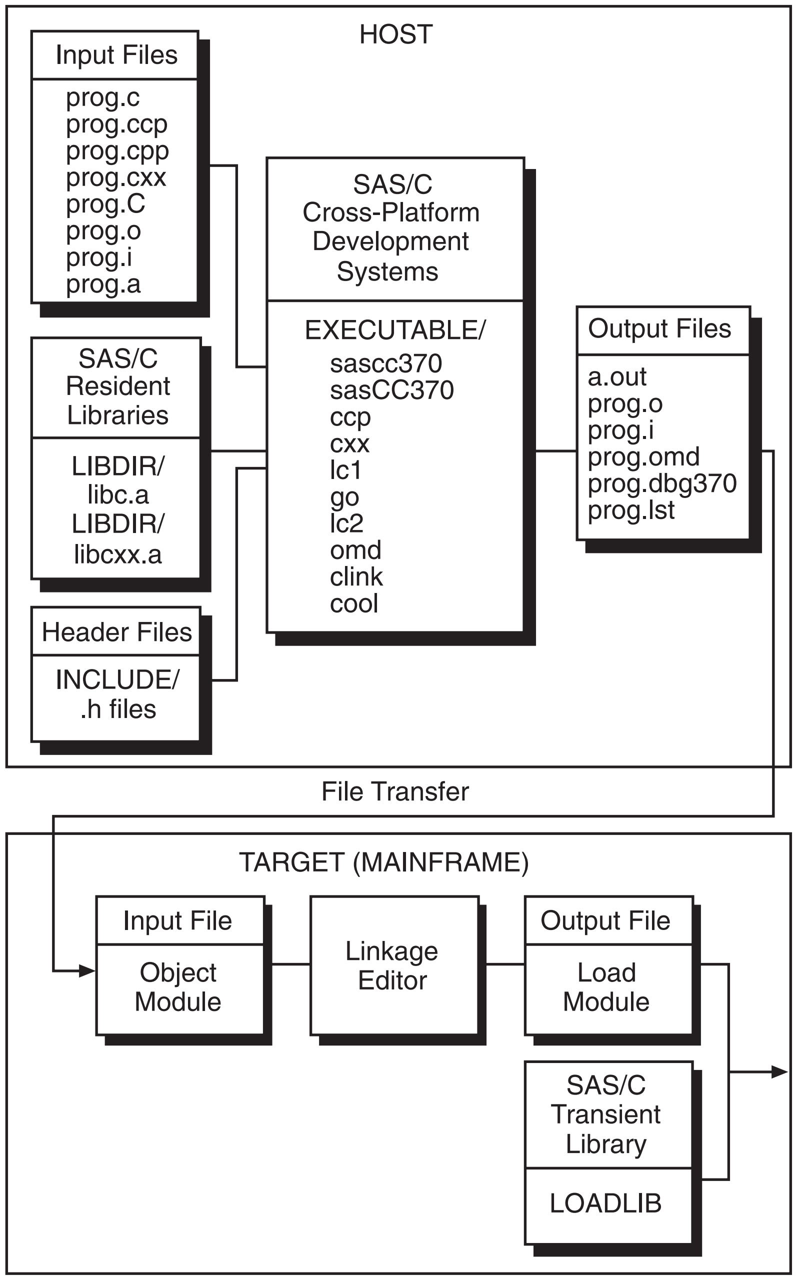
FTP uses the Transmission Control Protocol/Internet Protocol (TCP/IP) as the network mechanism for communicating between the host and target machines. Both FTP and TCP/IP are available at most mainframe sites. However, if this method of file transfer is not available at your site, you can use any file transfer method that enables you to transfer binary files between your workstation and the mainframe. The prelinked output file is already in binary form and does not require conversion from text to binary. Also, if you copy object files from the workstation to the mainframe, be sure the destination data set has a logical record length of 80 (LRECL=80) and a fixed record format (RECFM=F or FB).

Note: The SAS/C Connectivity Support Library (CSL) provides the capability of using Network File System (NFS) to copy files between the workstation and the mainframe using example utilities that are provided with the CSL product. See "Manipulating Files and Directories" on page 122 for additional information. △

SAS/C Cross-Platform Architecture

Figure 1.2 on page 6 illustrates the architecture of SAS/C cross-platform software by showing the executable files that compose the SAS/C Compiler and C++ Development System and its relationship to library, header, input, and output files. It also illustrates the relationship of the compiler to the host workstation and target mainframe.

Figure 1.2 Application Development Process



Executable Files

The SAS/C Cross-Platform system is composed of the executable files shown in Table 1.1 on page 7.

Table 1.1 Names of Executable Files

Functional Name	**Filename**
CICS command preprocessor	**`ccp`**
C compiler driver	**`sascc370`**
C++ driver	**`sasCC370`**
C++ translator	**`cxx`**
parser	**`lc1`**
code generator	**`lc2`**
global optimizer	**`go`**
object module dissembler	**`omd`**
`ar370` archive manager	**`ar370`**
prelinker (default)	**`cool`**
prelinker	**`clink`**

Note: There are several additional executable files for the utility programs provided with the SAS/C Cross-Platform Compiler. △

Executable Files Description

Like most compilers, the SAS/C Cross-Platform Compiler and C++ Development System performs the compilation in a series of phases.

The compiler drivers, **`sascc370`** and **`sasCC370`**, control the compilation, invoking the other executable files and passing them options during the various phases.

The CICS command preprocessor recognizes CICS commands embedded in your C and C++ source. The preprocessor translates these commands into appropriate function calls for communication with CICS.

The global optimizer, prelinker, and object module disassembler are enabled by compiler options and, like the parser and code generator, their execution is controlled by the compiler driver.

The global optimizer, **`go`**, performs advanced optimizations such as merging common subexpressions and eliminating code that is never executed, constant propagation, and strength reduction. The global optimizer also allocates registers, placing the most highly used variables for each section of code in registers. This eliminates any need for you to select and specify register variables.

The object module disassembler, **`omd`**, is a useful debugging tool that provides a copy of the assembler code generated for a C or C++ program. In addition to running the object module disassembler at compile time, it can also be invoked independent of the driver.

The prelinker, **`cool`**, is an object code preprocessor that merges CSECTs based on references to external variables. It provides the same functionality as the SAS/C **`COOL`** utility on the mainframe.

The C++ translator, **cxx**; parser, **lc1**; and code generator, **lc2**, are called by the compiler and C++ drivers to perform the actual compilation of the source code. Together, they handle the parsing, semantics analysis, instruction selection, and code emission phases of compilation.

The **ar370** archive utility is used to generate groups of files that are combined into a single archive file.

See Chapter 5, "Using the Global Optimizer and the Object Module Disassembler," on page 69 for more information about **go** and **omd**.

Library and Header Files

The SAS/C Cross-Platform Compiler and C++ Development System requires that the resident portions of the SAS/C and C++ libraries be located on the host workstation. The standard C and C++ header files must also be located on the host workstation. Optionally, you can specify additional header files as described in Chapter 2, "Using the SAS/C Cross-Platform Compiler and C++ Development System," on page 11. The transient portion of the SAS/C and C++ Libraries is located on the target mainframe.

Input and Output Files

Input to the SAS/C Cross-Platform Compiler and C++ Development System is provided as C source code, C++ source code, or previously compiled object files. The output is either in the form of unlinked or prelinked object files. Output files for the object module disassembler and the debugger may also be generated. The input and output files are described in greater detail in Chapter 2, "Using the SAS/C Cross-Platform Compiler and C++ Development System," on page 11.

Installation Considerations

The location of the executable files that compose the SAS/C cross-platform software is site-specific, and you should contact your on-site SAS/C Installation Representative if you are not sure where these files are located on your workstation.

man Pages

The unformatted **man** pages listed in Table 1.2 on page 8 are supplied with the SAS/C Cross-Platform Compiler and should be located in the **man1** subdirectory.

To use these **man** pages, you must add install_location to your **MANPATH** environment variable and install_location/**host**/hostname/**bin** to your **PATH** environment variable. The install_location and hostname are specific to your site.

Table 1.2 Available man Pages

man Page	Provides Information for ...
ar370	**ar370** archive utility
atoe	ASCII-to-EBCDIC translation utility
ccp	CICS command preprocessor

man Page	Provides Information for ...
`cool`	`cool` prelinker
`clink`	`clink` prelinker
`dset`	display license information
`etoa`	EBCDIC-to–ASCII translation utility
`mf2unix`	mainframe-to-UNIX C source code translation utility
`objdump`	SAS/C 370 object file dump utility
`omd`	describes the object module disassembler
`sascc370`	`sascc370` C compiler driver
`sasCC370`	`sasCC370` C++ driver
`spatch`	SAS/C binary patching application
`unix2mf`	UNIX-to-mainframe C source code translation utility
`update`	SAS/C update utility
`zap`	SAS/C zap, used with license information

Relationship to the Mainframe SAS/C C and C++ Development Systems

The SAS/C Cross-Platform Compiler and C++ Development System is a direct descendent of the mainframe system. Your mainframe C and C++ source code can be compiled by either product without modification.

Because of different maintenance structures, the SAS/C Cross-Platform Compiler and C++ Development System and the mainframe SAS/C C and C++ Development System do not always produce identical object code. However, the performance of applications compiled with either compiler will be essentially identical.

Debugging Considerations

To take complete advantage of a cross-development environment, it is preferable to retain the source code on the host workstation. This avoids the added burden of maintaining two copies of the source on two platforms.

You can use the SAS/C Debugger running on the mainframe to debug source code located on your workstation. The SAS/C Debugger has the ability to access source code and debugging information located on the workstation. The debugger uses the SAS/C CSL run-time transients that are provided with the SAS/C Cross-Platform Compiler to establish client/server connectivity between the workstation and the mainframe. This capability is described in Chapter 9, "Cross-Debugging," on page 93, Appendix 1, "Installing and Administering the NFS Client," on page 107, and Appendix 2, "Using the NFS Client," on page 117. SAS/C CSL provides many additional features, which are described in SAS Technical Report C-113, SAS/C Connectivity Support Library, Release 1.00.

CHAPTER
2

Using the SAS/C Cross-Platform Compiler and C++ Development System

Invoking the SAS/C and C++ Cross Platform Compiler

On a UNIX System

The SAS/C C and C++ cross-platform compiler is invoked in a manner similar to other compilers commonly used on UNIX platforms. The syntax for the commands used to invoke the compiler consists of the filename of the driver, followed by a list of options, and the filenames of the source code. This method of invoking the cross-platform compiler enables you to specify options using a syntax that is very comfortable to UNIX users, providing easy integration with UNIX build facilities. This method of compiling C and C++ programs is described in the section "Compiling C Programs under UNIX" on page 13 and "Compiling C++ Programs under UNIX" on page 14.

On a Windows System

The SAS/C C and C++ cross-platform compiler can be invoked either under the Microsoft-DOS shell or within the Microsoft Visual C++ IDE. The syntax for the commands used to invoke the compiler consists of the driver name, followed by a list of options, and the filenames of the source code. This method of invoking the cross-platform compiler enables you to specify options using a syntax that is familiar to PC users, providing easy integration with PC build facilities such as batch files or an integrated development environment.

The methods for compiling C and C++ programs in a DOS shell are described in "Using SAS/C C and C++ under a DOS Shell" on page 27 and in the Microsoft Developer Studio are described in "Using SAS/C C and C++ under the Microsoft Visual C++ IDE" on page 34.

CICS Command Preprocessor

The SAS/C CICS Command Preprocessor enables you to develop application programs to run under CICS. This application-programming interface enables you to request CICS services by placing CICS commands anywhere within your C or C++ source code. The SAS/C CICS preprocessor translates these commands into appropriate function calls for communication with CICS.

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

Compiling C Programs under UNIX

This section explains how to invoke the SAS/C Cross-Platform Compiler with the **sascc370** command.

Using sascc370

The **sascc370** compiler driver controls the compilation of your C source code. Invoke the compiler driver with the following command:

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

If specified, the options argument can be one or more of the compiler options described in Chapter 3, "Compiling C and C++ Programs," on page 39 or the **cool** options described in Chapter 6, "Prelinking C and C++ Programs," on page 73. You can also view a partial listing of these options online by issuing the **sascc370** command without any arguments. Some of the compiler options are particular to the **sascc370** driver, and others will alter the compilation in some manner. As mentioned in Chapter 1, "Overview of the SAS/C Cross-Platform Compiler and C++ Development System," on page 3, the compiler driver processes these options during the phases of compilation, passing them to the appropriate executable file as necessary.

If you do not specify any compiler options, the cross-platform compiler will generate prelinked, non-reentrant object code by default. Prelinking is accomplished by **cool**, which is normally invoked by the compiler driver.

It should also be noted that the cross-platform compiler generates object code targeted for an OS/390 environment by default. If you are compiling programs that you intend to run under CMS, you should specify either the **-Tcms370** or the **-Tpcms370** compiler option. See Chapter 3, "Compiling C and C++ Programs," on page 39 for more information about these options.

The filename arguments specify a list of input files that are to be compiled or prelinked. Files with a **.c**, **.C**, **.cpp**, or **.cxx** extension will be compiled (filenames that end with **.C**, **.cpp**, or **.cxx** are assumed to be C++ input files); files with a **.o** extension will be prelinked. See "Files" on page 19 for more information about the files used by the cross-platform compiler.

Note: The **-Kextname** option, which enables the use of extended filenames, is on automatically when you use **sascc370**. To disable the use of extended names, you must specify **-Knoextname**. △

Examples

The following examples are command line invocations of the cross-platform compiler using **sascc370**:

sascc370 alpha.c
Compiles the file **alpha.c**, generating the prelinked output file **a.out**. Notice that **a.out** is the default filename for prelinked output.

Normally, the prelinked output is copied to the mainframe for final linking. However, you can also copy the object files to the mainframe and use **COOL** to generate a load module. See "Linking C and C++ Programs" on page 17 for additional information.

sascc370 -c alpha.c
Compiles the file **alpha.c**, generating the object file **alpha.o**. The **-c** compiler option specifies that the object should not be prelinked.

sascc370 -o beta alpha.c
Compiles the file **alpha.c**, generating the prelinked output file **beta**. The **-o** option is used to specify the name of the prelinked output file. Notice that **beta** is generated instead of the default **a.out**.

sascc370 -o beta alpha.c gamma.cxx
Compiles the file **alpha.c** as a C source file and the file **gamma.cxx** as a C++ source file, generating the prelinked output file **beta** and the object file **alpha.o** and **gamma.o**. The **-o** option is used to specify the name of the prelinked output file. Notice that **beta** is generated instead of the default **a.out**.

sascc370 -Kextname alpha.c
Compiles **alpha.c**, which may contain external C identifiers of lengths greater than 8 characters. The **-Kextname** compiler option specifies extended names.

sascc370 -o gamma alpha.c beta.c
Compiles the files **alpha.c** and **beta.c**. The object files are then prelinked by **cool**, combining the output into the **gamma** file. The **gamma** file would then need to be copied to the mainframe for final linking.

sascc370 alpha.o beta.c
Compiles **beta.c**, which is then prelinked with **alpha.o** and the C libraries to produce the prelinked output file **a.out**.

sascc370 -Krent -Tcms370 alpha.c beta.c
Compiles the files **alpha.c** and **beta.c**, generating reentrant code targeted for a CMS system running under VM/ESA or VM/XA. The **-Krent** option specifies that reentrant modification of external variables is allowed, and the **-Tcms370** option specifies that the cross-platform compiler preprocessor should use predefined CMS symbols and link with the CMS libraries.

sascc370 -Tpcms370 -Tallres notrans.c
Compiles **notrans.c**, using the all-resident library to generate an all-resident program targeted for a CMS system running in System/370 mode (pre-bimodal).

sascc370 -Tspe sysprog.c
Compiles **sysprog.c**, using the SPE library to generate a program targeted for the C Systems Programming Environment under OS/390.

Compiling C++ Programs under UNIX

This section explains how to invoke the SAS/C Cross-Platform C++ Compiler directly with the **sasCC370** command.

Using sasCC370

The **sasCC370** compiler driver controls the compilation of your C++ source code. Invoke the compiler driver with the following command:

```
sasCC370 [options] [filename1 [filename2...]]
```

If specified, the options argument can be one or more of the compiler options described in Chapter 3, "Compiling C and C++ Programs," on page 39, or the **cool** options described in Chapter 6, "Prelinking C and C++ Programs," on page 73. (You can also view a partial listing of these options online by issuing the **sasCC370** command without any arguments.) Some of the compiler options are particular to the **sasCC370** driver, and others will alter the compilation in some manner. As mentioned in Chapter 1, "Overview of the SAS/C Cross-Platform Compiler and C++ Development System," on page 3, the compiler driver processes these options during the phases of compilation, passing them to the appropriate executable file as necessary.

If you do not specify any compiler options, the C++ development system will generate prelinked, non-reentrant object code by default. Prelinking is accomplished by **`cool`**, which is normally invoked by the compiler driver.

Note: The C++ development system generates object code targeted for an OS/390 environment by default. If you are compiling programs that you intend to run under CMS, you should specify either the **`-Tcms370`** or the **`-Tpcms370`** compiler option. See Chapter 3, "Compiling C and C++ Programs," on page 39 for more information about these options. △

The filename arguments specify a list of input files that are to be compiled or prelinked. Files with a **`.cxx`**, **`.cpp`**, **`.C`**, or **`.c`** extension will be considered to be C++ input and compiled as such. Files with a **`.o`** extension will be prelinked. See "Files" on page 19 for more information about the files used by the cross-platform compiler.

It should also be noted that the **`-Kextname`** option, which enables the use of extended names, is on automatically when you use **`sasCC370`**; extended names processing cannot be disabled for C++ compilations.

The **`sasCC370`** command is functionally equivalent to the **`sascc370`** command with the **`-cxx`** option.

Examples

The following examples are command line invocations of the cross-platform compiler using **`sasCC370`**:

`sasCC370 alpha.cxx`
: Translates and compiles the file **`alpha.cxx`**, generating the prelinked output file **`a.out`**. Notice that **`a.out`** is the default filename for prelinked output.

  Normally, the prelinked output is copied to the mainframe for final linking. However, you can also copy the object files to the mainframe and use **`COOL`** to generate a load module. See "Linking C and C++ Programs" on page 17 for more information about prelinking and linking.

`sasCC370 -c alpha.cxx`
: Translates and compiles the file **`alpha.cxx`**. The **`-c`** compiler option specifies that the object should not be prelinked; therefore, the output file **`a.out`** is not generated.

`sasCC370 -o beta alpha.cxx`
: Translates and compiles the file **`alpha.cxx`**, generating the prelinked output file **`beta`**. The **`-o`** option is used to specify the name of the prelinked output file. Notice that **`beta`** is generated instead of the default **`a.out`**.

`sasCC370 -o gamma alpha.cxx beta.cxx`
: Translates and compiles the files **`alpha.cxx`** and **`beta.cxx`**. The object files are then prelinked by **`cool`**, combining the output into the **`gamma`** file. The **`gamma`** file would then need to be copied to the mainframe for final linking.

`sasCC370 alpha.o beta.cxx`
: Translates and compiles **`beta.cxx`**, which is then prelinked with **`alpha.o`** and the C++ and C libraries to produce the prelinked output file **`a.out`**.

`sasCC370 -Krent -Tcms370 alpha.cxx beta.cxx`
: Translates and compiles the files **`alpha.cxx`** and **`beta.cxx`**, generating reentrant code targeted for a CMS system running under VM/ESA or VM/XA. The **`-Krent`** option specifies that reentrant modification of external variables is allowed, and

the **-Tcms370** option specifies that the cross-platform compiler preprocessor should use predefined CMS symbols and link with the CMS libraries.

Compiling C and C++ Programs under a DOS Shell

This section explains how to invoke the SAS/C Cross-Platform C and C++ Compiler directly with the **sascc370** command.

Under UNIX, you use the **sascc370** compiler driver to compile C object code and the **sasCC370** compiler driver to compile C++ object code. Under Microsoft-DOS, however, you use the **sascc370** compiler driver to compile both C and C++ object code.

Using sascc370

The **sascc370** compiler driver controls the compilation of your C or C++ object code. Invoke the compiler driver with the following command:

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

If specified, the options argument can be one or more of the compiler options described in Chapter 3, "Compiling C and C++ Programs," on page 39, or the **cool** options described in Chapter 6, "Prelinking C and C++ Programs," on page 73. You can also view a partial listing of these options online by issuing the **sascc370** command without any arguments. Some of the compiler options (for example, **-v**)are particular to the **sascc370** driver, and others will alter the compilation phases in some manner. The compiler driver processes these options during the phases of compilation, passing them to the appropriate executable file as necessary.

If you do not specify any compiler options, the cross-platform compiler will generate prelinked, non-reentrant object code by default. Prelinking is accomplished by **cool**, which is normally invoked by the compiler driver.

Note: The cross-platform compiler generates object code targeted for an OS/390 environment by default. If you are compiling programs that you intend to run on another operating system, you should specify the **-Txxx** option. For example, if you are compiling programs you intend to run under CMS, specify either the **-Tcms370** or the **-Tpcms370** compiler option. See Table 3.1 on page 42 for more information about these options. △

The filename arguments specify a list of input files that are to be compiled or prelinked. Files with a **.cxx**, **.cpp**, or **.c** extension are considered to be C++ input and are compiled as such. Files with a **.o** extension are prelinked. See "Files" on page 19 for more information about the files used by the cross-platform compiler.

The **sascc370** compiler driver examines the filename extension to determine whether the file contains source code (**.cxx**, **.cpp** , or **.c**) or compiled objects (**.o**, **.obj**, or **.a**). The driver then takes actions based on the compiler options and the types of input files specified on the command line.

Examples

The following examples are command line invocations of the cross-platform compiler using **sascc370**:

sascc370 alpha.c
Compiles the file **alpha.c**, generating the prelinked output file **a.out**. Notice that **a.out** is the default filename for prelinked output. Normally, the prelinked output is copied to the mainframe for final linking.

`sascc370 -c alpha.c`
Compiles the file **`alpha.c`**, generating the prelinked output file **`a.out`**. The **`-c`** compiler option specifies that the object should not be prelinked.

`sascc370 -o beta alpha.c`
Compiles the file **`alpha.c`**, generating the prelinked output file **`beta`**. The **`-o`** option is used to specify the name of the prelinked output file. Notice that **`beta`** is generated instead of the default **`a.out`**.

`sascc370 -o beta alpha.cxx`
Translates and compiles the file **`alpha.cxx`**, generating the prelinked output file **`beta`**. The **`-o`** option is used to specify the name of the prelinked output file. Notice that **`beta`** is generated instead of the default **`a.out`**.

`sascc370 -o beta alpha.c gamma.cxx`
Compiles the source file **`alpha.c`** as a C source file and **`gamma.cxx`** as a C++ source file, generating the object files **`alpha.o`** and **`gamma.o`**. The object files are then prelinked by **`cool`**, combining the output into the **`beta`** file. The **`-o`** option is used to specify the name of the prelinked output file. Notice that **`beta`** is generated instead of the default **`a.out`**.

`sascc370 -Kextname -Krent alpha.c`
Compiles **`alpha.c`**, which may contain external C identifiers of lengths greater than 8 characters. The **`-Kextname`** compiler option specifies extended names. The **`-Krent`** option specifies that reentrant modification of external variables is allowed. The prelinked output file will be named **`a.out`**.

Linking C and C++ Programs

Linking in the cross-platform environment usually involves prelinking on the host system and then copying the prelinked file to the target system, either OS/390 or CMS, where the final linking occurs. Prelinking is performed by **`cool`**, which is normally called when you compile your source code. Final linking of the prelinked output on the mainframe can be accomplished with the IBM linkage editor or CMS **`LOAD`** and **`GENMOD`** commands.

Although prelinking is performed by default when you call the cross-platform compiler, you can suppress prelinking by specifying the **`-c`** compiler option. Note, however, that prelinking is required, either on the workstation or on the mainframe, if any of the following conditions are true:

- More than one compilation initializes an **`__rent`** variable. There are four ways a variable is assigned the **`__rent`** attribute:
  1. The variable is external and the compiler option **`-Krent`** or **`-Krentext`** is used.
  2. The variable is static and the compiler option **`-Krent`** is used.
  3. The variable is external and the name begins with an underscore.
  4. The variable is declared **`__rent`**.
- The **`-Kextname`** option is specified for more than one compilation.
- The **`-Tcms370`** or **`-Tpcms370`** options are specified and the cumulative length of the pseudoregisters exceeds the maximum size allowed by the CMS 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.

If you do not call the prelinker during compilation, you can perform the prelinking in one of the following ways:

- □ Call **cool** directly
- □ Copy the object files to the mainframe and use **COOL** to perform the prelinking.

This section discusses each of these methods of prelinking.

Using cool

The prelinker, **cool**, is an executable file that can be called directly. The following is the syntax for invoking **cool**:

```
cool [options] [filename1 [filename2 ...]]
```

The options argument specifies any of the prelinker options described in Chapter 6, "Prelinking C and C++ Programs," on page 73. You must use the **-o** option to specify an output file when you invoke **cool**. If you enter the **cool** command without specifying the **-o** option, an error message is displayed along with a partial listing of the options accepted by **cool**.

The filename arguments specify a list of input files that are to be prelinked. You must specify the complete name of the file; **cool** does not assume a **.o** extension.

The SAS/C C library objects are located in the **ar370** archive /libdir/**libc.a**. The libdir depends on where the product was installed and which target you are compiling for. The SAS/C C++ objects are located in the **ar370** archive /libdir/**libcxx.a**. In order to resolve references to SAS/C C and C++ library functions, these **ar370** archives must be included in the **cool** command. If you are prelinking a C++ program, you must specify /libdir/**libcxx.a** before specifying /libdir/**libc.a**.

Examples

The following examples are command line invocations of **cool**:

cool -o prog alpha.o beta.o /libdir/**libc.a**

Prelink the object files **alpha.o** and **beta.o** to produce the prelinked output file **prog**.

libc.a must be included in order to resolve references to SAS/C C library functions. The directory specification, libdir, for this library is target and site-specific. If you are not sure where it is located at your site, compile a program with the **-v (verbose)** option to see the default command line used by the **sascc370** or **sasCC370** compiler driver to invoke **cool**.

cool -o prog myojb.o /libdir/**libcxx.a** /libdir/**libc.a**

In this example, a C++ object file, **myojb.o**, is being prelinked, generating the prelinked output file **prog**.

Note: If you are using **cool** to prelink a C++ program, you must specify /libdir/**libcxx.a** before specifying /libdir/**libc.a**. The SAS/C C++ library functions must be resolved before SAS/C C library functions. The directory specification, libdir, for this library is site-specific. If you are not sure where it is located at your site, compile a program with the **-v (verbose)** option to see the default command line used by the **sascc370** or **sasCC370** compiler driver to invoke **cool**. △

cool -o prog interface.o io_handler.o /libdir/**libares.a**

In this example, the all-resident library, **libares.a**, is specified. Specialized applications, such as all-resident or SPE applications, must be linked with special library routines. See "Library Files" on page 20 and the SAS/C Compiler and Library User's Guide for more information.

cool -o prog -w binary_tree.o /libdir/**libc.a**
Prelink the object file **binary_tree.o**, generating the prelinked output file **prog**. The **-w** option specifies that warning messages should be suppressed.

Using COOL on the Mainframe

The other method of prelinking your object files is to copy them to the mainframe where they can be linked with the **COOL** object code utility. The SAS/C Compiler and Library User's Guide describes the **COOL CLIST** and the **COOL EXEC**, which can be used to invoke **COOL** on the mainframe.

Note: You can use the UNIX **cat** command or the DOS **copy /b** command to combine multiple object files into a single file on the workstation, copy that file to the mainframe, and then run **COOL** on the mainframe on the copied file. This is often easier than copying each file individually. △

Also note that if you use extended names or the **-Aprem** option, you should not prelink both on the workstation and on the mainframe.

Files

As described in "Executable Files" on page 7, the cross-platform compiler actually consists of executable files. The **sascc370** and **sasCC370** compiler drivers are responsible for processing the options you specify on the command line and controlling the compilation of C and C++ programs. The other executable files are the translator, parser, global optimizer, code generator, prelinker, and the object module disassembler. The location of these files is site-specific.
The cross-platform compiler also supplies and uses additional files that are located in other directories. These files are described in Table 2.1 on page 19.

Table 2.1 Additional Files Required

Files	Location
Library	The resident portion of the SAS/C C and C++ libraries is located in the **ar370** archives found in the **lib** directory. Since the transient portion of the SAS/C C and C++ libraries contains routines that must be loaded during program execution, the transient portion must be located on the mainframe with your executable load module. Different **ar370** archives are provided for different mainframe target environments.
Header	The standard SAS/C C and C++ header files are located in an **include** directory. Additional directory locations for include files can be specified.
Input	Your source code, compiled objects, or **ar370** archive input files can be located in any directory you choose.
Output	Your compiled or prelinked output can be directed to files in any directory you choose.

The next four sections of this chapter provide additional information about the library, include, input, and output files used by the cross-platform compiler.

Library Files

The SAS/C C and C++ libraries contain both resident and transient routines. (They also provide all-resident and SPE routines that can be used in specialized applications.) Resident routines are incorporated into your program before it is executed. Transient routines are dynamically loaded during program execution.

Resident Library Routines

The resident libraries, **`libc.a`** for C and **`libcxx.a`** for C++, contain routines that are prelinked with your application. Because these routines are added to your program during prelinking, resident library routines are not dynamically loaded during program execution. The resident library is specific to your target: OS/390, CMS (VM/ESA Mode), or CMS (System/370 mode).

Transient Library Routines

The transient library is a collection of system-dependent routines that are loaded as needed by a program during execution. 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.

All-resident Library Routines

In most programming situations, the dynamic loading and unloading of routines from the transient library 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 are used. In order to create an all-resident program, your program must include routines from the all-resident library.

The **`-Tallres`** compiler option is used to specify an all-resident program. You can also invoke **`cool`** directly, specifying the location of the all-resident library, libdir/ **`libares.a`**, as an argument to **`cool`**.

SPE Library Routines

The SAS/C SPE library, **`libspe.a`**, provides resident routines that support the C Systems Programming Environment (SPE). The **`-Tspe`** compiler option can be used to specify an SPE program. You can invoke **`cool`** directly, specifying the location of the SPE library as an argument to **`cool`**.

Header Files

C and C++ source files take advantage of preprocessor support for including auxiliary source in a compilation by using the **`#include`** mechanism. The cross-platform compiler

C and C++ preprocessors interpret **`#include`** statements that cause auxiliary files to be included as part of the source being compiled.

Auxiliary source files that are not provided as part of the SAS/C C and C++ libraries are considered to be user header files, and are typically enclosed in double quotes. For example

```
#include "error_messages.h"
```

Auxiliary files that are provided with the SAS/C C and C++ libraries are called library header files or system header files, and are typically enclosed in angle brackets. For example

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

The library header files are installed with the product in the **`include`** subdirectory on UNIX and in the ***`installdir`*****`\include`** subdirectory on Windows 95 and Windows NT. This location is called the system directory on all three platforms.

When the SAS/C C and C++ preprocessors encounter a **`#include`** statement, the auxiliary file to be included must be located. The compilers search for the named file in a way that is typical of UNIX compilers. To modify the search order, use the **`-Knousearch`** option. The following describes the search performed by the preprocessor.

- If the filename in the **`#include`** statement is a complete pathname, beginning from the root directory (/), then no other searching is performed. If the C or C++ system is unable to read the named file, an error is produced.
- If the file was enclosed in double quotes and the **`-Knousearch`** option was not specified, the search proceeds as follows. Look in:
  1. The directory of the source file containing the **`#include`** statement.
  2. Any directories specified by the **`-I`** compiler option. This search is done in the order that the **`-I`** options appear on the command line.
  3. The system directory.
- If the file was enclosed in angle brackets and the **`-Knousearch`** option was not specified, the search proceeds as follows. Look in:
  1. Any directories specified by the **`-I`** compiler option. This search is done in the order that the **`-I`** options appear on the command line.
  2. The system directory.
- If the file was enclosed in double quotes and the **`-Knousearch`** option was specified, the SAS/C C preprocessors take the following steps to locate the named file. Look in:
  1. The current working directory.
  2. Any directories specified by the **`-I`** compiler option. This search is done in the order that the **`-I`** options appear on the command line.
  3. The directory of the source file containing the **`#include`** statement.
  4. The system directory.
- If the file was enclosed in angle brackets and the **`-Knousearch`** option was specified, the SAS/C C preprocessors take the following steps to locate the named file. Look in:
  1. The current working directory.
  2. The system directory.

The search order described in this section is traditional with UNIX C compilers, and therefore meshes well with many of the programming tools commonly used under the UNIX operating system, such as **`make`**.

Adding Directories to the Search Path under UNIX

The **-I** compiler option in the SAS/C Cross-Platform Compiler enables you to specify additional directories that are searched before the standard include-file search list. For example, the following command causes the cross-platform compiler to search the **/u/userid/branch_bank/headers** directory for header files:

```
sascc370 -I/u/userid/branch_bank/headers
   debits.c credits.c
```

In this example, **debits.c** or **credits.c** could contain the statement

```
#include "transactions.h"
```

which would be located in the **/u/userid/branch_bank/headers** directory. The **-I** option works the same for **sasCC370**.

Adding Directories to the Search Path under a Windows Environment

When the SAS/C C and C++ preprocessors encounter a **#include** statement, the auxiliary file to be included must be located. The compiler searches the SASCINCLUDE environment variable, which was automatically defined at installation. You have two options for adding additional user directories to the search path. Depending on your desired search order, you can either prefix or append the SASCINCLUDE environment variable in your ***installdir*\host\wnt\bin\sascc.cfg** configuration file. Or, you can use the **-I** compiler option in the SAS/C Cross-Platform Compiler to enable you to specify additional directories that are searched before the standard include–file search list. For example, either of the following solutions causes the cross-platform compiler to first search the **c:\user\branch_bank\headers** directory, then the system directory, for header files:

```
sascc370 -Ic:\user\branch_bank\headers
   debits.c credits.c
```

Or, you could change the **sascc.cfg** file to explicitly prefix the user header file directory to the INCLUDE statement:

```
INCLUDE=c:\user\branch_bank\headers;
   %SASCINCLUDE%
```

In these two examples, **debits.c** and **credits.c** could contain the statement:

```
#include "transactions.h"
```

which would be located in the **c:\user\branch_bank\headers** directory.

Input Files

Under a Windows environment, the **sascc370** compiler driver accepts C source, C++ source, and previously compiled objects. Under UNIX the the **sascc370** compiler driver accepts C source and previously compiled objects, and the **sasCC370** compiler driver accepts C++ source and previously compiled objects. However, the **sascc370** compiler driver can accept C++ source under UNIX if you specify **-cxx** on the **sascc370** command line.

sascc370 Input Files

The **sascc370** compiler driver accepts the following input files.

Table 2.2 sascc370 Input Files

Input Files	Description
C Source Files	uncompiled C source code, identified by a **`.c`** filename extension.
C++ Source Files	uncompiled C++ source code, identified by **`.cpp`**, **`.cxx`**, or **`.C`**.
Compiled Object Files	contain previously compiled object code. Identified by a **`.o`** filename extension.
Archive Libraries	**`ar370`** archive files. Identified by a **`.a`** filename extension.

The **`sascc370`** compiler driver examines the filename extension to determine whether the file contains source code (**`.c`**) or compiled objects (**`.o`** or **`.a`**). The driver then takes action based on the compiler options and the types of input files specified on the command line. For a **`.c`** file, the following actions are taken:

1 Invoke the cross-platform compiler to produce a **`.o`** object file.
2 Invoke **`cool`** with the system **`ar370`** archive, **`libc.a`**, and the **`.o`** object file to produce a prelinked object file.

The output files produced by this sequence are described in “Output Files” on page 23.

sasCC370 Input Files

The **`sasCC370`** compiler driver accepts the following input files.

Table 2.3 sasCC370 Input Files

Input Files	Description
C++ Source Files	uncompiled C++ source code, identified by a **`.cpp`**, **`.cxx`**, **`.C`**, or **`.c`** filename extension.
Compiled Object Files	contain previously compiled object code. Identified by a **`.o`** filename extension.
Archive Libraries	**`ar370`** archive files. Identified by a **`.a`** filename extension.

The **`sasCC370`** compiler driver examines the filename extension to determine whether the file contains source code (**`.cpp`**, **`.cxx`**, **`.C`**, or **`.c`**) or compiled objects (**`.o`** or **`.a`**). The driver then takes action based on the compiler options and the types of input files specified on the command line. For a **`.c`** file, the following actions are taken:

1 Invoke the cross-platform compiler to produce a **`.o`** object file.
2 Invoke **`cool`** with the system **`ar370`** archive, **`libc.a`**, and the **`.o`** object file to produce a prelinked object file.

The output files produced by this sequence are described in “Output Files” on page 23.

Output Files

Depending on how they are invoked, the **`sascc370`** and **`sasCC370`** compiler drivers produce any of the following output file types:

Table 2.4 sascc370 Output Files

Output Files	Contents
Object Files	unlinked object code and are identified by a **.o** filename extension.
Prelinked Output Files	object code that has been prelinked by **cool**. By default, the prelinked output is written to **a.out**. The **-o** compiler option is used to direct the output to another file.
Preprocessed Source Files	source code that has been preprocessed and are identified by a **.i** extension. Preprocessed source code has all the macros and **#include** files expanded. These files are generated by invoking the **sascc370** compiler driver with the **-P** option.
Debugger Files	information used by the SAS/C Debugger and are identified by a **.dbg370** filename extension. These files are produced if the **-g** compiler option is specified.
OMD Output Files	output from the Object Module Disassembler. These files are identified by a **.omd** filename extension. See Chapter 5, "Using the Global Optimizer and the Object Module Disassembler," on page 69 for more information.
Listing Files	output listings and are identified by a **.lst** filename extension. The **-Klisting** compiler option is used to specify a particular listing file.

The prelinked object file can be copied to the mainframe, where it is then submitted to the linkage editor, which accomplishes the final linking and generates an executable module. Note that your output files will be targeted for an OS/390 environment by default. Use either the **-Tcms370** or the **-Tpcms370** compiler options to generate output files that are compatible with the CMS environment. Use the **-Tcics** compiler option to generate output files that are compatible with the CICS environment.

The **.dbg370** debugger files are required to debug your program with the SAS/C Debugger. See Chapter 9, "Cross-Debugging," on page 93 for more information about using the SAS/C Debugger in the cross-platform development environment.

Output Filename Generation

Unless you use the **-o** compiler option to specify an output filename, the base filename of the source file will be used to generate the base filenames of the output object and listing files. For example, suppose you invoked the cross-platform compiler with the following command:

```
sascc370 -Kilist students.c
```

In this example, the **-Kilist** option specifies that a header file listing should be generated. The **students.c** file contains uncompiled source code that includes a header file that will be printed to the output listing. The following output files are produced:

Table 2.5 Example Output Files

File	Description
`a.out`	prelinked output file
`students.lst`	listing file containing the source code for the included header file.
`students.o`	compiled file containing object code.

Using -o with a Single Source File

If file **`students.c`** is compiled with the **`-o`** option, the output object and listing filenames are formed with the specified base name. For example, suppose the cross-platform compiler is invoked as follows:

```
sascc370 -o roster -Kilist students.c
```

In this case, the following output files are generated:

Table 2.6 Example Output Files

File	Description
`roster`	prelinked output file
`students.lst`	listing file containing the source code for the included header file.
`students.o`	compiled file containing object code.

Notice that the **`a.out`** file is not generated in this case; instead, the prelinked object is written to the file specified with **`-o`**.

Using -o with Multiple Source Files

With multiple input files, the base name of each source file is used to generate the base of the **`.o`** and **`.lst`** filenames associated with the source file. For example,

```
sascc370 -o acct_bal -Kilist debit.c credit.c
```

In this case, the following output files are generated:

Table 2.7 Example Output Files

File	Description
`acct_bal`	prelinked output file
`debit.lst`	listing file containing the source code for the header file included by **`debit.c`**.
`credit.lst`	listing file containing the source code for the header file included by **`credit.c`**.

File	Description
credit.o	compiled file containing object code.
debit.o	compiled file containing object code.

Windows Environment Configuration File

A configuration file named **sascc.cfg** was created when you installed the SAS/C C and C++ driver on your PC. This file contains important location information used by the **sascc370** driver. If you move the software after the initial installation, or if you need to permanently define additional **#include** search paths, you need to modify **sascc.cfg**.

Following is an example **sascc.cfg** data file:

```
#
#  PATH points to the <installation dir>
#  INCLUDE points to <installation dir>\include
#
#  The Environment variables SASCDEV and SASCINCLUDE
#  are generated at the time of installation of the
#  SAS/C and C++ Cross-Platform product. Only change
#  these two variables if you alter the location of
#  the installation on your PC.
#
#  Use the 'set' command in an MS-DOS shell to change
#  the Environment variables, or concatenate
#  explicit pathname qualifiers to the following
#  PATH and INCLUDE variables.
#
PATH=%SASCDEV%
INCLUDE=%SASCINCLUDE%
```

sascc.cfg is found in the ***installdir*\host\wnt\bin** directory. The **sascc370** compiler driver uses the SASCDEV and SASCINCLUDE environment variables to determine the installation directory, and thus determine the locations of the components listed in Table 2.8 on page 26.

Table 2.8 Component Paths

Component	Location
Executables	PATH\host\wnt\bin
Standard library directory	PATH\lib
System include files	INCLUDE

The SASCDEV and SASCINCLUDE environment variables were automatically set during the install process. PATH and INCLUDE direct the **sascc370** driver to the appropriate installation directory and the location of the system header files. The PATH and INCLUDE identifiers in the **sascc.cfg** configuration file are independent of the Windows PATH and INCLUDE environment variables set in the Windows operating system.

Using SAS/C C and C++ under a DOS Shell

The following sections provide examples of different DOS shell batch files used to generate compiled object code and prelinked output for final execution on the mainframe. See "Using SAS/C C and C++ under the Microsoft Visual C++ IDE" on page 34 for a description of similar objectives under Microsoft Visual C++ IDE.

Compiling C and C++ Source Code

The following sample compile batch file will accept as input a **.c** file and compile the code to produce only an object deck (**-c**), allowing reentrant modification of static and external data (**-Krent**), and defining a section name as the source code filename (**-Ksname**). (Note the section name must be seven characters or less.) The **-v** option specifies that both the driver messages and the command lines that execute each phase of the cross-platform compiler are echoed to the **%LOG%** file. If the source file is not in the current working directory, the quoted, qualified pathname should be entered as the second command-line argument.

```
@echo off
set NAME=%1
set PATHNAME=
if NOT '%2'=='' set PATHNAME=%2\
set SOURCE=%PATHNAME%%NAME%.c
set OBJECT=%NAME%.o
set LOG=%NAME%.clg
set C_OPTS=-c -v -Krent -Ksname=%NAME%
erase %LOG%
erase %OBJECT%
sascc370 %C_OPTS%  %SOURCE% -o %OBJECT% > %LOG%
echo Done with %NAME%.
```

Following is the correct syntax to invoke this sample compile.bat file:

```
compile sourcename [pathname]
```

For example, to compile the file **D:\Program Files\sasc\samples\c\ftoc.c** and produce **ftoc.o** in the current working directory, enter the following command:

```
compile ftoc "d:\Program Files\sasc\samples\c"
```

where **ftoc** on the command line is the source code filename without the **.c** extension.

At installation, the PATH environment variable is prefixed to include the location of the **sascc370** driver (***installdir*\host\wnt\bin**), so the driver name in these batch files should not require an explicit pathname.

Prelinking Object Code

The following sample prelink batch file accepts as input the previously compiled object code filename (without the **.o** extension) and produces prelinked output from **cool**. The **-v** option specifies that any driver messages, and the command lines that execute **cool**, are echoed to the **%LOG%** file. If the object file is not in the current working directory, the quoted, qualified pathname should be entered as the second command-line argument.

```
@echo off
set NAME=%1
set PATHNAME=
if NOT '%2'=='' set PATHNAME=%2\
set OBJECT=%PATHNAME%%NAME%.o
set OUTPUT=%NAME%
set LOG=%NAME%.llg
set L_OPTS=-v
erase %LOG%
erase %OUTPUT%
sascc370 %L_OPTS%  %OBJECT% -o %OUTPUT% > %LOG%
echo Done with %NAME%.
```

Following is the correct syntax to invoke this sample **link.bat** file:

```
link objectname [pathname]
```

For example, to prelink the object file **ftoc.o** and generate **ftoc**, enter the following command:

```
link ftoc
```

where **ftoc** on the command line is the object code filename without the **.o** extension.

Building Source Code

Alternatively, to compile and prelink source code in one step, you could execute the following batch file. The **sascc370** driver executes the SAS/C C and C++ Cross-Platform compiler, which produces prelinked output generated by **cool.** In the following example, the compiler options specify reentrant code, extended names processing (**-Kextname**), a section name, run-time type identification (**-Krtti**), and instantiation of class templates (**-Kautoinst**). A quoted include pathname is added for user-defined header files. The command line that executes each phase of the cross-platform compiler is displayed in the **%LOG%** file.

```
@echo off
set NAME=%1
set PATHNAME=
if NOT '%2'=='' set PATHNAME=%2\
set OUTPUT=%NAME%
set LOG=%NAME%.log
set BLD_OPTS=-v -Krent -Kextname -Ksname=%NAME%
    -Krtti -Kautoinst
set INCL= -I"d:\Program Files\sasc\samples\h"
erase %LOG%
erase %OUTPUT%
sascc370 %BLD_OPTS% %INCL% %SOURCE% -o %OUTPUT% > %LOG%
echo Done with %NAME%.
```

Following is the correct syntax to invoke this sample **build.bat** file:

```
build sourcename [pathname]
```

For example, to compile and prelink the file

```
d:\Program Files\sasc\samples\cxx\tsttmpl.cxx
```

enter the following command:

```
build.bat tsttmpl "d:\Program Files\sasc\samples\cxx"
```

where **tsttmpl** on the command line is the C++ source code filename without the **.cxx** extension. This method produces two files. In this example, the compiler generates the object file **tsttmpl.o** and the prelinker produces the prelinked output file **tsttmpl**.

The output **tsttmpl.log** file should look something like the example in Output 2.1 on page 29:

Output 2.1 Example log file

```
SAS/C Compiler Driver V6.50.01
Copyright (C) 1998 SAS Institute Inc.
set INCLUDE370='d:\Program Files\SASC\Include'
"d:\Program Files\SASC\host\wnt\bin\cxx"  -Adigraph1 -Adigraph2
-DCROSS370=1 -XA -Hu '-r' '-Arrti' "-Id:\Program Files\Sasc\Samples\h"
'-MC:\TEMP\sascca00344.1.ai' '-mtsttmpl'
"d:\program files\sasc\samples\cxx\tsttmpl.cxx"
"C:\TEMP\sascca00344.1.c"
"d:\Program Files\SASC\host\wnt\bin\lc1" -dCROSS370=1 -cd -hu -n! '-r'
'-n!' "-id:\Program Files\Sasc\Samples\h" -cxx -d__CXX_PRIMARY__=1
-q011=1 '-ststtmpl' -xc
"-oC:\TEMP\sascca00344.1.q" "C:\TEMP\sascca00344.1.c
SAS/C Release 6.50.01 (Target 370 Cross Compiler)
Copyright (c) 1998 by SAS Institute Inc. All Rights Reserved.
*** No errors; No warnings; No user suppressed warnings
"d:\Program Files\SASC\host\wnt\bin\lc2" "-oC:\TEMP\sascca00344.1.o"
"C:\TEMP\sascca00344.1.q"
SAS/C Compiler (Phase 2) 6.50.01
Copyright (c) 1998 by SAS Institute Inc. All Rights Reserved.

"d:\Program Files\SASC\host\wnt\bin\lc1" -dCROSS370=1 -cd -hu -n! '-r'
'-n!' "-id:\Program Files\Sasc\Samples\h" -cxx -d__CXX_SECONDARY_0__=1
-q011=2 '-q012=@%TEMPL' -xc "-oC:\TEMP\sascca00344.1.q"
"C:\TEMP\sascca00344.1.c"
SAS/C Release 6.50.01 (Target 370 Cross Compiler)
Copyright (c) 1998 by SAS Institute Inc. All Rights Reserved.
*** No errors; No warnings; No user suppressed warnings
"d:\Program Files\SASC\host\wnt\bin\lc2" "-oC:\TEMP\sascca00344.2.o"
"C:\TEMP\sascca00344.1.q"
SAS/C Compiler (Phase 2) 6.50.01
Copyright (c) 1998 by SAS Institute Inc. All Rights Reserved.

"d:\Program Files\SASC\host\wnt\bin\lc1" -dCROSS370=1 -cd -hu -n! '-r'
'-n!' "-id:\Program Files\Sasc\Samples\h" -cxx -d__CXX_SECONDARY_1__=1
-q011=3 '-q012=@%TEMPL' -xc "-oC:\TEMP\sascca00344.1.q"
"C:\TEMP\sascca00344.1.c"
SAS/C Release 6.50.01 (Target 370 Cross Compiler)
Copyright (c) 1998 by SAS Institute Inc. All Rights Reserved.
*** No errors; No warnings; No user suppressed warnings
"d:\Program Files\SASC\host\wnt\bin\lc2" "-oC:\TEMP\sascca00344.3.o"
"C:\TEMP\sascca00344.1.q"
SAS/C Compiler (Phase 2) 6.50.01
Copyright (c) 1998 by SAS Institute Inc. All Rights Reserved.

"d:\Program Files\SASC\host\wnt\bin\sheller" -c -o tsttmpl.o
"@C:\TEMP\sascca00344.1.shell"
"d:\Program Files\SASC\host\wnt\bin\cool" -o "tsttmpl"
-L"d:\Program Files\SASC" tsttmpl.o
"d:\Program Files\SASC"\lib\libcxx.a
"d:\Program Files\SASC"\lib\mvs\libc.a
SAS/C (R) C Object code Pre-Linker Release 6.50.01
Copyright (c) 1998 by SAS Institute Inc. All Rights Reserved.

cool: Note 1010: Pre-Linking completed with return code = 0
```

Customizing Microsoft Visual C++ Integrated Development Environment Menus

The following information applies only to Version 5.0 and later of the Microsoft Visual C++ Integrated Development Environment (IDE).

If you selected the option to integrate SAS/C into the Microsoft Visual C++ IDE, two additional menu items, **`Compiler Options`** and **`SAS/C Help Files`**, appear on the

Tools pull-down menu in the Microsoft Visual C++ IDE after you install SAS/C on your PC. You can permanently move these items to a more intuitive area of the menu bar in the Microsoft Visual C++ IDE by importing one of the following files into your Registry:

layout50.reg
: if you are using Microsoft Visual C++ IDE, Version 5

layout60.reg
: if you are using Microsoft Visual C++ IDE, Version 6

Importing a Layout File

The following procedure describes how to use the Windows Start button on the taskbar to import the layoutnn.reg file for the version of the Microsoft Visual C++ IDE you are using:

1. Select **Run** from the Start button pop-up menu.
2. Type **regedit** in the **Open** text box.
3. Select **Registry**.
4. Select **Import Registry File...**.
5. Select the *installdir*\host\\wnt\\bin directory in the Import Registry File.
6. Select the layout*nn*.reg file for the version of the Microsoft Visual C++ IDE you are using.
7. Click **Open**.

Adding SAS/C Compiler Options and Help Files to a Pull-Down Menu

The following two procedures describe how to add the SAS/C **Compiler Options** and the **SAS/C Help Files** dialogs to the pull-down menus of the Microsoft IDE. When you launch a new session of the Microsoft IDE, a new pull-down menu item named **SAS/C** will appear. The **SAS/C** menu item enables access to the **Compiler Options** and the **SAS/C Help Files** dialogs. You can use either procedure.

- Use the following procedure if you prefer to leave the **Compiler Options** item in the **Tools** pull-down menu and copy the **SAS/C Help Files** item to the **Help** pull-down menu:
  1. Select **Customize** from the **Tools** pull-down menu.
  2. Select the Commands tab.
  3. Select **Tools** from the **Category** list.
  4. Select and drag **user-defined tool 8** over the **Help** pull-down menu.
  5. Drop the **tool 8** item onto the most convenient area of the **Help** pull-down menu.
  6. Close the Customize dialog box.
- Use the following procedure if you prefer to create a new pull-down menu on the toolbar titled **SAS/C** and copy the **Compiler Options** and **SAS/C Help Files** items to this menu:
  1. Select **Customize** from the **Tools** pull-down menu.
  2. Select the **Commands** tab.
  3. Select **new menu** from the **Category** list.
  4. Drag and drop the the **new menu** item from the Command list box to an empty spot on the toolbar docking area.

5 Right-click on the new menu item.

6 Select **`Button Appearance...`** to open a dialog box that enables the **`Button Text Update`**.

7 Enter **`SAS/C`** in the appropriate text box.

8 Select a **`Category`** for adding menu items, for example, **`Tools`**.

9 Select **`tool 7`** in the **`Command`** list box.

10 Drag and drop **`tool 7`** on the new menu, **`Custom Menu #`**.

11 Select **`tool 8`** in the **`Command`** list box.

12 Drag and drop **`tool 8`** on the new menu, **`Custom Menu #`**.

13 Close the **`Customize`** dialog box.

14 Open **`Customize`** from the **`Tools`** pull-down menu again.

15 Select **`Menus`** from the **`Category`** list.

16 Select the new menu, **`Custom Menu #`** from the **`Command`** list box.

17 Drag and drop **`Custom Menu #`** on the **`Menu`** bar.

Now you can delete the Menu item from the toolbar.

Using the SAS/C and C++ Cross-Platform Compiler under the Microsoft Visual C++ IDE

If you prefer to use the Microsoft Developer Studio, you can use the SAS/C and C++ Cross-Platform Compiler within the Microsoft Visual C++ Integrated Development Environment. To ensure that the SAS/C and C++ Cross-Platform Compiler is invoked in the Microsoft Developer Studio, follow these steps:

1 Select the **`Tools`** menu from the Microsoft Developer Studio toolbar.

2 Select **`Compiler Options`**.

3 Select **`SAS/C and C++ Cross-Platform Compiler`**.

4 Select **`OK`**.

When you select the SAS/C and C++ Cross-Platform Compiler from the Microsoft Developer Studio **`Tools`** menu, it becomes the default compiler whenever you restart Microsoft Visual C++. The item for the SAS/C and C++ Cross-Platform Compiler on the Compiler Options menu is the default selection, but it does not reflect the current compiler selection set up in the Registry.

To compile and prelink source code using the SAS/C and C++ Cross-Platform Compiler, you first need to configure the compile and prelink options in the Win32 Release settings for your project. Although some menu differences exist between Version 4.2 and later versions of Microsoft Visual C++, these operations apply to Version 4.2 and later versions. The following examples are taken from Microsoft Visual C++ 5.0, but the SAS/C++ Cross-Platform Compiler is compatible with Version 4.2 and later of Microsoft Visual C++.

Configuring Compile and Prelink Options

The following example uses a project named **`ftoc`**. Use the following procedure to configure the compile and prelink options for the **`ftoc`** project in Microsoft Visual C++:

1 Select the **`Build`** menu.

2 Select **`Set Active Configuration`**.
3 Select the **`ftoc-Win32 Release`** project configuration.
4 Select the **`Project`** menu.
5 Select the **`Settings`** option.
6 Select the C/C++ tab.
7 Remove all the Microsoft Visual C++ compiler options except for the following: **`/ML`**.

The SAS/C and C Cross-Platform Compiler ignores the **`/ML`** option. Also, the Microsoft Visual C++ compiler options in Table 2.9 on page 33 are interpreted as valid SAS/C Cross-Platform Compiler options.

Table 2.9 Recognized Microsoft Visual C++ Compiler Options

Microsoft Option	**Description**
/Fd*filename*	Renames program database file
/nologo	Suppresses display of sign-on banner
/Dname[=def]	Defines constants and macros
/c	Compiles without linking
/Zi	Generates complete debugging information
/O	Executes Global Optimizer
/O1	Executes Global Optimizer
/O2	Executes Global Optimizer
/Idirectory	Searches a directory for include files

In the **`Project Options`** field, you can add any of the SAS/C and C++ Cross-Platform Compiler options that are listed in the compiler options list in the SAS/C Cross-Platform Compiler and C++ Development System: Usage and Reference. For example, to specify verbose commands, automatic instantiation of class templates, run-time type identification, and re-entrant code generation, add the **`-v`**, **`-Kautoinst`**, **`-Krtti`**, and **`-Krent`** options for **`sascc370`**.

Delete all the project options under the Project Settings Link tab except for **`/incremental:no`**, **`/pdb:"/ftoc.pdb"`**, and **`/machine:IX86`**. The **`sascc370`** driver ignores these three options. Specify the name of the prelinked output file in the **`Output file name`** field. Add any **`sascc370`** driver options for prelinking, such as **`-v`**, to the **`Project Options`** field. The **`Project Options`** field should appear as **`/incremental:no /pdb:"/ftoc.pdb" /machine:IX86 /out:"ftoc" -v`**.

Adding and Deleting Compiler and Project Options

You can use the Project Options window to add any SAS/C and C++ Cross-Platform Compiler options. Similarly, you can use the Project Settings Link tab to delete all the Project Options except **`/incremental:no/pdb:''/ftoc.pdb''/machine:IX86`**. These options are enforced by the Microsoft Visual C++ IDE, but they are ignored by the SAS/C and C++ Cross-Platform Compiler.

If source-level settings are required for a project, you can use the following procedure to set options for each source file in the Settings For text box in the Project Settings dialog box:

1 Select the C/C++ tab.
2 Select **Category**.
3 Select **General** from the drop-down menu and enter your preprocessor definitions.
4 Select **Category** again.
5 Select **Preprocessor** and enter any additional include directories.
6 Select the **Undefined symbols** text box, and enter any **sascc370** options for the compiler that are to be allocated to the selected source file.

Note: These driver options are not defined by the Microsoft Visual C++ IDE, and are applicable only to **sascc370**. A **/U** will prefix any options entered in this area when you view it in the Source File Options text box. The **sascc370** driver ignores the **/U**, and it passes any associated options to the appropriate phases of compilation.
To compile and pre-link source code, select **Build ftoc** from the **Build** pull-down menu. △

Compiling and Prelinking Object Code

To compile and prelink the source code, select the **Build ftoc** item from the **Build** menu. The SAS/C and C++ Cross-Platform Compiler generates a compiled object file, **ftoc.obj**, and a prelinked output file, **ftoc.exe**. Any execution messages or error diagnostics are displayed in the Output view.

Using SAS/C C and C++ under the Microsoft Visual C++ IDE

If you prefer to use the Microsoft Developer Studio, you can use SAS/C C and C++ within the Microsoft Visual C++ Integrated Development Environment. To ensure the SAS/C and C++ compiler is invoked in the Microsoft Developer Studio,

1 Select the Tools menu from the Microsoft Developer Studio toolbar.
2 Select Compiler Options.
3 Select Cross-Platform SAS.
4 Select OK.

When you select the SAS/C and C++ compiler from the Microsoft Developer Studio Tools menu, it becomes the default compiler whenever you restart Microsoft Visual C++. The selection button for the SAS/C and C++ compiler on the Compiler Options menu will not be highlighted, but the SAS/C and C++ compiler is the default compiler. If you later change the compiler selection from the SAS/C and C++ compiler to the Microsoft compiler, the Microsoft compiler becomes the default compiler.
To compile and prelink source code using the SAS/C and C++ compiler, you first need to configure the compile and prelink options in the Win32 Release settings for your project. Although some menu differences exist between Microsoft Visual C++ Version 4.2 and Version 5.0, these operations apply to both versions. The following examples are taken from Microsoft Visual C++ 5.0, but the SAS/C C and C++ compiler is compatible with Versions 4.2 and 5.0 of Microsoft Visual C++.

Configuring Compile and Prelink Options

The following example uses a project named **ftoc**. Use the following procedure to configure the compile and prelink options for the **ftoc** project in Microsoft Visual C++:

1 Select the **`Build`** menu.
2 Select **`Set Active Configuration`**.
3 Select the **`ftoc-Win32 Release`** project configuration.
4 Select the **`Project`** menu.
5 Select the **`Settings`** option.
6 Select the **`C/C++`** tab.
7 Remove all the Microsoft Visual C++ compiler options except for the following: **`/ML /FO"Release/"`**.

The SAS/C C and C++ compiler ignores the **`/ML`** and **`/FO"Release/"`** options. Also, the Microsoft Visual C++ compiler options in Table 2.10 on page 35 are interpreted as valid SAS/C cross-platform compiler options.

Table 2.10 Recognized Microsoft Visual C++ Compiler Options

Microsoft Option	Description
/Fd*filename*	Renames program database file
/nologo	Suppresses display of sign-on banner
/Dname[=def]	Defines constants and macros
/c	Compiles without linking
/Zi	Generates complete debugging information
/O	Executes Global Optimizer
/O1	Executes Global Optimizer
/O2	Executes Global Optimizer
/Uname	Eliminates initial name definition
/Idirectory	Searches a directory for include files

In the Project Options field, you can add any of the SAS/C C and C++ compiler options that are listed in Chapter 3, "Compiling C and C++ Programs," on page 39. For example, to specify verbose commands, automatic instantiation of class templates, run-time type identification, and reentrant code generation, add the **`-v`**, **`-Kautoinst`**, **`-Krtti`**, and **`-Krent`** options for **`sascc370`**.

Delete all the project options under the Project Settings Link tab except for **`/incremental:no`**, **`/pdb:"Release/ftoc.pdb"`**, and **`/machine:IX86`**. The **`sascc370`** driver ignores these three options. Specify the names of the input object modules in the **`Object/library modules`** field. Specify the name of the prelinked output file in the **`Output file name`** field. In this example, although the input object filename, **`ftoc.o`**, is located in directory **`D:\Program Files\DevStudio\MyProjects\temperature`**, the **`Object/library modules`** field does not contain the fully qualified object file pathname. Add any **`sascc370`** driver options for prelinking, such as **`-v`**, to the Project Options field. The **`Project Options`** field should appear as **`ftoc.o /incremental:no /pdb:"Release/ftoc.pdb" /machine:IX86 /out:"ftoc" -v`**.

Compiling and Prelinking Object Code

To compile and prelink the source code, select the **`Build ftoc`** item from the **`Build`** menu. SAS/C C and C++ generate a compiled object file, **`ftoc.o`**, and a prelinked output file, **ftoc**. Any execution messages or error diagnostics are displayed in the Output view. For this example, when building **ftoc.cxx**, the output window should contain something like the output in Output 2.2 on page 36:

Output 2.2 Example output window contents

```
- - - - - - - - - - Configuration: ftoc - Win32 Release - - - - - - - - - -

Compiling...
SAS/C Compiler Driver V6.50.01
Copyright (C) 1998 SAS Institute Inc.
set INCLUDE370='d:\Program Files\SASC\Include'
"d:\Program Files\SASC\host\wnt\bin\cxx" -Adigraph1 -Adigraph2
-DCROSS370=1 -XA -Hu '-r' '-Arrti' '-MC:\TEMP\sascca00319.1.ai'
'mftoc' "D:\Program Files\Dev\Studio\MyProjects\temperature\ftoc.cxx"
"C:\TEMP\sascca00319.1.c"
sascc370: Invalid '/FoRelease' ignored
SAS/C C++ 6.50.01 (Mar 2 1998)
Copyright (C) 1998 SAS Institute Inc.
"d:\Program Files\SASC\host\wnt\bin\lc1" -dCROSS370=1 -cd -hu -n! '-r'
-cxx -d__CXX_PRIMARY__=1 -q011=1 '-sftoc' -xc
"-oC:\TEMP\sascca00344.1.q" "C:\TEMP\sascca00319.1.c"
SAS/C Release 6.50.01 (Target 370 Cross Compiler)
Copyright (c) 1998 by SAS Institute Inc.  All Rights Reserved.
*** No errors; No warnings; No user suppressed warnings
"d:\Program Files\SASC\host\wnt\bin\lc2" "-oC:\TEMP\sascca00319.1.o"
"C:\TEMP\sascca00319.1.q"
SAS/C Compiler (Phase 2) Release 6.50.01
Copyright (c) 1998 by SAS Institute Inc.  All Rights Reserved.
"d:\Program Files\SASC\host\wnt\bin\sheller" -c -o ftoc.o
@C:\TEMP\sascca00319.1.shell
You have selected the SAS C/C++ Cross Platform Compiler
Linking...
SAS/C Compiler Driver 6.50.01
Copyright (c) 1998 SAS Institute Inc.
set INCLUDE370='d:\Program Files\SASC\Include'
"d:\Program Files\SASC\host\wnt\bin\cool" -o "ftoc"
-L"d:\Program Files\SASC" ftoc.o "d:\Program Files\SASC"\lib\libcxx.a
"d:\Program Files\SASC"\lib\mvs\libc.a
SAS/C (R) C Object code Pre-Linker Release 6.50.01
Copyright (c) 1998 by SAS Institute Inc.  All Rights Reserved.
cool: Note 1010: Pre-Linking completed with return code = 0
You have selected the SAS C/C++ Cross Platform Linker

ftoc - 0 error(s) 0 warning(s)
```

ar370 Archives

An **ar370** archive library is a collection of object files, similar to a partitioned data set under OS/390 or a TXTLIB under CMS. The advantage of **ar370** archives is that they combine several files into one file, and they maintain a list of definitions of variables and functions. The items in the list are not limited in length, which allows references to long symbols to be resolved during linking. Furthermore, collecting many objects together in one **ar370** archive can provide a single file for managing these objects.

Note: External references to variables are resolved by extracting files that define the reference from the **ar370** archive. This is similar to the autocall process under OS/390 or CMS; however, there is an important distinction. On OS/390 and CMS, a reference is resolved by examining the names of the members of the partitioned data set or TXTLIB. Under UNIX, **cool** determines which file contains a defining instance of a reference by examining the ar370 generated symbol table, which is part of the **ar370** archive. Thus, references are resolved following a mechanism that is more common to UNIX implementations.

The **`ar370`** utility used to create and manage **`ar370`** archives is described in Chapter 7, “ar370 Archive Utility,” on page 85. △

CHAPTER 3

Compiling C and C++ Programs

Introduction

The SAS/C C and C++ cross-platform compiler accepts a number of options that allow 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.

Since the global optimizer and object module disassembler (OMD) are often executed as part of compilation, the options accepted by the global optimizer and OMD are also discussed in this chapter. The global optimizer and OMD are also discussed in Chapter 5, "Using the Global Optimizer and the Object Module Disassembler," on page 69.

Note: Several compiler options have new names in this release. For backward compatibility with previous releases, you can also use the old names. For details, see Appendix 4, "Redistributing SAS/C Product Files," on page 135. △

Syntax

As described in Chapter 2, "Using the SAS/C Cross-Platform Compiler and C++ Development System," on page 11, the basic syntax for invoking the **sascc370** compiler driver to compile your C object code is as follows:

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

The basic syntax for invoking the **sasCC370** compiler driver to compile your C++ object code is as follows:

```
sasCC370 [options] [filename1 [filename2...]]
```

The options argument for **sascc370** and **sasCC370** can be one or more of the driver options described in the section "Option Descriptions" on page 47. You can also specify **cool** options when you invoke the compiler driver, which are described in Chapter 6, "Prelinking C and C++ Programs," on page 73, and CICS preprocessor options, which are described in Chapter 4, "Using the SAS/C CICS Command Preprocessor," on page 65.

Specifying Phase of Compilation of C and C++ Programs

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.

Note: Though short-form options may resemble the OpenEditon shell options, they are often different. △

Note: For more information on long-form and short-form compiler options, see the chapter about compiling C programs in the SAS/C Compiler and Library User's Guide. △

The compilation of a C or C++ program with the cross-platform compiler occurs in the following phases:

1. CICS pre-processing
2. C++ parsing
3. C parsing
4. Optimization
5. Code generation
6. Prelinking

Some of the options passed to the **sascc370** or **sasCC370** compiler driver apply only to one of these phases. To indicate the particular phase of compilation, you must use the following syntax when specifying these options:

```
-Wphase,option
```

Note: There is no space after the comma between the phase and the option specifications. △

The phase can be any one of the following:

- **-W** followed by the letter **P** specifies that the option should be passed to the CICS command pre-processor.
- **-W** followed by the letter **C** specifies that the option should be passed to the C++ translation phase of the compilation.
- **-W** followed by the number **1** specifies that the option should be passed to the C parser phase of the compilation.
- **-W** followed by the letter **g** specifies that the option should be passed to the global optimizer.
- **-W** followed by the number **2** specifies that the option should be passed to the code generation phase of the compilation.
- **-W** followed by the letter *l* specifies that the option should be passed to the prelinker.

Note: This book uses italics to help you distinguish between the letter l and the number **1**. △

Examples

The following examples illustrate how the compilation phase is specified:

-Wg,-a
The **-Wg** specification indicates that the **-a** option should be passed to the global optimization phase of the compilation. The **-a** option specifies that the global optimizer should assume worst-case aliasing.

Note: All of the global optimizer options described in this chapter can also be passed to the compiler driver without specifying the compilation phase. For example, to pass the **-a** option directly, specify **-Oa**. △

-WP,-d
The **-WP** specification indicates that the **-d** option should be passed to the CICS command preprocessor. The **-d** option is described in Chapter 4, "Using the SAS/C CICS Command Preprocessor," on page 65.

Cross-Platform Compiler Defaults

The SAS/C Cross-Platform Compiler has several options that are specified by default. Most options that begin with **-K** have both a positive and negative form. Most of these default to their negative form if unspecified, but a few default to their positive form.

For example, if you do not specify the **-Kat** option, the compiler will not allow the use of the call-by-reference operator **@**; specifying **-Knoat** has the same effect. Options such as **-Knoextname** behave the opposite way. By default, the compiler processes extended names (**-Kextname**). To disable this feature, you must specify the **-Knoextname** option. Table 3.1 on page 42 lists the default for each compiler option with a positive and negative form.

All other compiler options default to their negative form if unspecified. For example, if you do not specify the **-g** option, the compiler does not generate a **.dbg370** debugging information file.

If you do not specify any compiler options, the cross-compiler generates prelinked, non-reentrant object code, by default. Note that there is no compiler option to specify prelinking; the **-c** option is provided to turn prelinking off. Use the **-Krent** or **-Krentext** options to enable reentrant modification of external data.

Option Summary

The cross-platform compiler options are summarized in Table 3.1 on page 42. A more detailed description of each option is provided in the section "Option Descriptions" on page 47.

The option specifications are listed in the first column of the table. The second 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. Only options that begin with **-K** can be negated. All other options cannot be negated. To negate a **-K** option, precede the option name with **no**. For example, to negate the **-Kasciiout** option, specify **-Knoasciiout**. The third column lists the default for each option that can be negated.

Table 3.1 Compiler Options

Option	Negation	Default	Description
-c	+		Suppress prelinking.
-cf	+		For C compilations: Require function prototypes in scope.
-cxx	+		Specifies to **sascc370** that **.c** files should be interpreted as C++ programs. The C++ library will be included in all linking.
-D*sym*[=*val*]	+		Defines the symbol sym as having the value specified by value during the preprocessing phase.
-g	+		Generate debuggable code and produce a **.dbg370** debugging information file. (See **-Kdebug**.)
-Gf*n*	+		Specify the maximum number of floating-point registers (n= 0 through 2). *Note:* **-Gf*n*** and **-Kfreg** are synonyms. △
-Gv*n*	+		Specify the maximum number of registers that the optimizer can assign to register variables (n= 0 through 6). *Note:* **-Gv*n***and **-Kgreg** are synonyms. △
-I*pathname*	+		Append pathname to the list of directories searched for include files.
-Kalias	!	**-Knoalias**	Specifies that the global optimizer should assume worst-case aliasing. *Note:* **-Oa** and **-Kalias** are synonyms. △
-Karmode	!	**-Knoarmode**	Specifies that code that uses the ESA access registers may be generated.
-Kasciiout	!	**-Knoasciiout**	Character string constants are output as ASCII values.
-Kat	!	**-Knoat**	Allow the use of the call-by-reference operator **@**.
-Kautoinst	!	**-Knoautoinst**	Controls automatic implicit instantiation of template functions and static data members of template classes.
-Kbitfield=*n*	+		For C compilations: allows for fields that are not **int**. Sets the allocation unit size for **int** to be n. For C++ compilations: sets the allocation unit size for **int** to be n. C++ always allows non-**int** bitfields.
-Kbytealign	!	**-Knobytealign**	Align all data on byte boundaries.
-Kcomnest	!	**-Knocomnest**	For C compilations: Allow nested comments.
-Kcomplexity=*n*	+		Specify the maximum complexity that a function can have and remain eligible for default inlining. *Note:* **-Oic** and **-Kcomplexity** are synonyms. △

Option	Negation	Default	Description
-Kdbgmacro	!	**-Knodbgmacro**	Specify that macro names should be saved in the debugger file.
-Kdbgobj	!	**-Knodbgobj**	Causes the compiler to place the debugging information in the output object file.
-Kdbhook	!	**-Knodbhook**	Generate debugger hooks.
-Kdebug [=*dbg370-filename*]	!	**-Knodebug**	Generate debuggable code and produce a **.dbg370** debugging information file. Optionally, specify the name of the debugging information file. *Note:* **-Kdebug** and **-g** are similar. **-Kdebug** allows a filename argument. △
-Kdepth=*n*	+		Specify the maximum depth of functions to be inlined. *Note:* **-Oid** and **-Kdepth** are synonyms. △
-Kdigraph	!	see description	Enables the translation of the International Standard Organization (ISO) digraphs and/or the SAS/C digraph extension **scd**.
-Kdollars	!	**-Knodollars**	Allow the use of the **$** character in identifiers, except as the first character.
-Kexcept	!	**-Knoexcept**	Enables exception handling for C++ code.
-Kexclude	!	**-Knoexclude**	For C compilations: Omit listing lines that are excluded by preprocessor statements from the formatted source listing.
-Kfreg=*n*	+		Specify the maximum number of floating-point registers (n=0 through 2). *Note:* **-Gf*n*** and **-Kfreg** are synonyms. △
-Kgreg=*n*	+		Specify the maximum number of registers that the optimizer can assign to register variables (n=0 through 6). *Note:* **-Gvn** and **-Kgreg** are synonyms. △
-Khlist	!	**-Knohlist**	For C compilations: Print standard header files in the formatted source listing.
-Kigline	!	**-Knoigline**	Ignore **#line** statements in the input file.
-Kilist	!	**-Knoilist**	For C compilations: Print the source referenced by the **#include** statement in the formatted source listing.
-Kindep	!	**-Knoindep**	Generate code that can be called before the run-time library framework is initialized or code that can be used for interlanguage communication.
-Kjapan	!	**-Knojapan**	Translates keywords and identifiers that are in uppercase to lowercase before they are processed by the compiler.
-Klisting [=*list-filename*]	!	**-Knolisting**	For C compilations: Generate a listing file and, optionally, specify the listing file name.
-Kloop	!	**-Knoloop**	Specify that the global optimizer should perform loop optimizations. (See **-Ol**.)

Option	Negation	Default	Description
-Kmaclist	!	**-Knomaclist**	For C compilations: Print macro expansions in the formatted source listing.
-Knarrow	!	**-Knonarrow**	For C compilations: Make the listing more narrow.
-Knodbgcmprs	!	**-Kdbgcmprs**	Do not compress debugging information file.
-Knoextname	!	**-Kextname**	Disable the use of extended names.
-Knohmulti	!	**-Khmulti**	For C compilations: Specifies that system include files will only be included once.
-Knoimulti	!	**-Kimulti**	For C compilations: Specifies that local include files will only be included once.
-Knoinline	!	**-Kinline**	Disable all inlining during the optimization phase. (See **Oin**.)
-Knoinlocal	!	**-Kinlocal**	Disable inlining of single-call, static functions during the optimization phase. (See **-Oil**.)
-Knolineno	!	**-Klineno**	Disable identification of source lines in run-time messages emitted by the SAS/C Library.
-Knostringdup	!	**-Kstringdup**	Create a single copy of identical string constants.
-Knousearch	!	**-Kusearch**	Specify **#include** file search rules that are not typical of UNIX.
-Komd[=*omd-filename*]	!	**-Knoomd**	Invoke the object module disassembler and, optionally, specify the **.omd** listing file name. *Note:* **-Komd** and **-S** are similar. **-Komd** allows a filename argument. △
-Koptimize	!	**-Knooptimize**	Execute the global optimizer phase of the compiler. *Note:* **-O** and **-Koptimize** are synonyms. △
-Koverload	!	**-Knooverload**	For C++ compilations: Turn on recognition of the overload C++ keyword.
-Kpagesize=*nn*	+		For C compilations: Specify the number of lines per page for source and cross-reference listings.
-Kpflocal	!	**-Knopflocal**	Assume that all functions are **_ _local** unless **_ _remote** was explicitly specified in the declaration.
-Kposix	!	**-Knoposix**	Create a POSIX-compliant program.
-Kppix	!	**-Knoppix**	For C compilations: Allow nonstandard token-pasting.
-Krdepth=n	+		Specifies the maximum level of recursion to be inlined. *Note:* **-Oir** and **-Krdepth** are synonyms. △
-Kredef	!	**-Knoredef**	Allow redefinition and stacking of **#define** names.
-Krefdef	!	**-Knorefdef**	Force the use of the strict reference-definition model for external linkage of **_ _rent** identifiers.
-Krent	!	**-Knorent**	Support reentrant modification of static and external data.
-Krentext	!	**-Knorentext**	Support reentrant modification of external data.

Option	Negation	Default	Description
-Krtti	!	**-Knortti**	Enables the generation of information required for RTTI on class objects that have virtual functions.
-Ksingleret	!	**-Knosingleret**	Forces the code generator to generate a single return sequence at the end of each function.
-Ksmpxivec	!	**-Knosmpxivec**	Generate a CSECT with a unique name of the form sname@. in place of @EXTVEC# (for SMP support).
-Ksname=*sname*	+		Define sname as the SNAME for a compilation.
-Ksource	!	**-Knosource**	For C compilations: Output a formatted source listing of the program to the listing file.
-Ksrcis= *source-filename*	+		Override the name of the source file in the debugging information (**.dbg370**) file.
-Kstrict	!	**-Knostrict**	For C compilations: Enable an extra set of warning messages for questionable or nonportable code.
-Ktmplfunc	!	**-Ktmplfunc**	Controls whether a nontemplate function declaration that has the same type as a template specialization refers to the template specialization. When **-Knotmplfunc** is specified, template specializations may also be referred to by nontemplate declarations. **-Knotmplfunc** provides compatibility with older code. **-Ktmplfunc** is the default for compatibility with the ISO C++ Standard.
-Ktrigraphs	!	**-Knotrigraphs**	For C compilations: Enable translation of ANSI standard trigraphs.
-Kundef	!	**-Knoundef**	Undefine predefined macros. *Note:* **-U** and **-Kundef** are synonyms. △
-Kuse_clink	!	**-Knouse_clink**	Use the **clink** program instead of **cool** to prelink the object file.
-Kvstring	!	**-Knovstring**	Generate character string literals with a 2-byte length prefix.
-Kxref	!	**-Knoxref**	For C compilations: Produce a cross-reference listing.
-Kzapmin=n	+		Specify the minimum size of the patch area, in bytes.
-Kzapspace=n	+		Change the size, n, of the patch area generated by the compiler.
-mrc	+		Use mainframe return code values instead of UNIX-style values.
-O	+		Execute the global optimizer phase of the compiler. *Note:* **-O** and **-Koptimize** are synonyms. △
-Oa	+		Specifies that the global optimizer should assume worst-case aliasing. *Note:* **-Oa** and **-Kalias** are synonyms. △

Option	Negation	Default	Description
-Oic=*n*	+		Specify the maximum complexity that a function can have and remain eligible for default inlining. *Note:* **-Oic** and **-Kcomplexity** are synonyms. △
-Oid=*n*	+		Specify the maximum depth of functions to be inlined. *Note:* **-Oid** and **-Kdepth** are synonyms. △
-Oil	+		Enables inlining of single-call, static functions during the optimization phase. Note: **-Oil** and **-Kinlocal** are synonyms.
-Oin	+		Enables inlining of small static and external functions during the optimization phase. (Functions defined with the **_ _inline** keyword are inlined by default.) *Note:* **-Oin** and **-Kinline** are synonyms. △
-Oir=n	+		Specifies the maximum level of recursion to be inlined. *Note:* **-Oir** and **-Krdepth** are synonyms. △
-Ol	+		Specify that the global optimizer should perform loop optimizations. *Note:* **-Ol** and **-Kloop** are synonyms. △
-o filename	+		Specifies the output filename.
-P	+		Only run the preprocessor on any **.c** files, generating **.i files**.
-Qpathname	+		Specify an alternative pathname to be searched for the cross-platform compiler executable file
-S	+		Invoke the object module disassembler after a successful compilation. (See **-Komd**.)
-Tallres	+		Specify that all-resident library routines should be used to build an all-resident program. *Note:* The **-Tspe** option is not allowed in combination with the **-Tallres** option. △
-Tcics370	+		Specify that CICS is the target.
-Tcicsvse	+		Specify that CICS running under the VSE operating system is the target. *Note:* The **-Tspe** option is not allowed in combination with the **-Tcicsvse** option. △
-Tcms370	+		Specify CMS running under VM/ESA or VM/XA as the target host operating system.
-Tpcms370	+		Specify CMS running in System/370 mode (pre-bimodal) as the target host operating system.

Option	Negation	Default	Description
-Tspe	+		Specify that the SAS/C SPE library routines should be used to build an SPE program. *Note:* The **-Tallres** and **-Tcicsvse** options are not allowed in combination with the **-Tspe** option. △
-temp=*directory*	+		Specify a different temporary directory for the compiler to use.
-U	+		Undefine predefined macros. *Note:* **-U** and **-Kundef** are synonyms. △
-v	+		Specify verbose mode.
-w~*n*	+		Cause warning message n to be treated as an error condition.
-w+*n*	+		Specify that warning number n should not be suppressed.
-w*n*	+		Suppress warning message number n.

Option Descriptions

This section provides a more detailed description of each of the options that were summarized in Table 3.1 on page 42. Unless otherwise specified, the options apply to both C and C++ source files.

-c
suppresses prelinking. By default, **sascc370** and **sasCC370** will invoke the prelinker after the compilation is complete. The **-c** option can be used to suppress this default action.

-cf
requires that all functions and function pointers have a prototype in scope. If the **-cf** 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.

Note: The **-cf** option is equivalent to the SAS/C Compiler **reqproto** option for C compilations only. △

-cxx
specifies to **sascc370** that **.c** files should be interpreted as C++ source files. Also, the C++ library will be used in all linking. This causes **sascc370** to be functionally equivalent to **sasCC370**.

-D*sym*[=*val*]
defines a symbol, sym, and assigns an optional value, val.
The **-D** option is equivalent to the SAS/C Compiler define option.

-g
allows the use of the debugger to trace the execution of statements at run time. (The compiler produces debugging information that is written to the **.dbg370** file.) For programs not compiled with **-g**, only calls can be traced.

Note: If you use **-g**, the **-l** option, which enables the identification of source lines in run-time messages, is implied. Also note that the **-g** option causes the compiler to suppress all optimizations as well as store and fetch variables to or from memory more often. △

The **-g** option is equivalent to the SAS/C Compiler and C++ Development Systems debug option. (See **-Kdebug**.)

-Gf*n*
: specifies the maximum number, n, of floating point registers that the optimizer can assign to register variables in a function. The n argument can have a value of 0 through 2, inclusive (the default is 2). The **-Gf** option is used only with optimization (specified by the **-O** option).

The **-Gf** option is equivalent to the SAS/C Compiler **freg** option. (See **-Kfreg**.)

-Gv*n*
: specifies the maximum number, n, of registers that the optimizer can assign to register variables in a function. **-Gv** is used with **-O** only. The n argument is 0 to 6, inclusive (the default is 6).

The **-Gv** option is equivalent to the SAS/C Compiler **greg** option. (See **-Kgreg**.)

-I*pathname*
: appends the specified pathname to the lists of directories searched for include files. See "Header Files" on page 20 for more information about the search path used by the cross-platform compiler.

-Kalias
: is a synonym for the **-Oa** option.

-Karmode
: specifies that code that uses the ESA access register may be generated. This option is required to compile code that uses far pointers. See the section on far pointer support in the SAS/C Compiler and Library User's Guidefor more information on the **armode** option, which is the host system version of the **-Karmode** option.

-Kasciiout
: causes character string constants to be output as ASCII values. By default, character string constants are output as EBCDIC values.

The **-Kasciiout** option is equivalent to the SAS/C Compiler **asciiout** option.

-Kat
: allows the use of the call-by-reference operator @.

The **-Kat** option is equivalent to the SAS/C Compiler and C++ Development Systems **at** option.

-Kautoinst
: The **-Kautoinst** 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 **-Kautoinst** 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, Second Edition, Release 6.50 for information about templates and automatic instantiation.)

Note: COOL must be used if this option is specified. △

-Kbitfield=*n*
: specifies the size of the allocation unit for **int** bitfields. This option requires that you specify a value, n. The values can be either 1, 2, or 4, which specifies that the allocation unit be a **char**, **short**, or **long**, respectively.

Note: For C source files, this option allows non-**int** bitfields. For C++ source files, non-**int** bitfields are always allowed. Refer to SAS/C Compiler and Library User's Guide, Fourth Edition for more details. △

The **-Kbitfield** option is equivalent to the SAS/C Compiler and C++ Development Systems **bitfield** option.

-Kbytealign

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 System/370 conventions, even with the **-Kbytealign** 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 using byte alignment.

The **-Kbytealign** option is equivalent to the SAS/C Compiler and C++ Development Systems **bytealign** option.

-Kcomnest

allows nested comments.

The **-Kcomnest** option is equivalent to the SAS/C Compiler comnest option. For C compilations only.

-Kcomplexity=n

is a synonym for the **-Oic** option.

-Kdbgmacro

specifies that definitions of macro names should be saved in the **.dbg370** debugger file.

Note: This substantially increases the size of the file. △

The **-Kdbgmacro** option is equivalent to the SAS/C Compiler **dbgmacro** option.

-Kdbgobj

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 **-Kautoinst** option, **-Kdbgobj** is enabled automatically.

By default, the **-Kdbgobj** option is off. The short form for the option is **-xc**. See the SAS/C C++ Development System User's Guide, Second Edition, Release 6.50 for information about templates and automatic instantiation.

Note: COOL must be used if this option is specified. △

-Kdbhook

generates hooks in the object code. When you compile a module with the **-g** option, the **-Kdbhook** option is implied. **-Kdbhook** can be used with the **-O** option to enable debugging of optimized object code.

The **-Kdbhook** option is equivalent to the SAS/C Compiler **dbhook** option.

-Kdebug[=*dbg370-filename*]
generates debuggable code and produces a **.dbg370** debugging information file. Optionally, you can specify the name of the debugging information file with the **-Kdebug=*dbg370-filename*** option.

The **-Kdebug** option is similar to the **-g** option. When you specify **-Kdebug=*dbg370-filename***, **-g** is assumed.

-Kdepth=*n*
is a synonym for the **-Oid** option.

-Kdigraph
Digraph options enable the translation of the International Standard Organization (ISO) digraphs and 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 digraph support offered collectively by ISO and SAS/C. Table 3.2 on page 50 gives a brief description of the new digraph compiler options.

Table 3.2 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 3.3 on page 50 provides the default values and an example of how to negate the options in each of the different environments.

Table 3.3 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 3.4 on page 51 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.

Table 3.4 ISO digraph Alternative Tokens

Rel 6.50 Tokens	
Primary	**Alternate**
{	<%
}	%>
[	<:
]	:>
#	%:
##	%:%:

Note: See the chapter about special character support in the SAS/C Compiler and Library User's Guide for more information on digraphs. △

-Kdollars
allows the use of the **$** character in identifiers, except as the first character.
The **-Kdollars** option is equivalent to the SAS/C Compiler **dollars** option.

-Kexcept
Enables code generation for exception handling in the C++ translator. This option is not enabled by default because it can add additional overhead to the generated code. If exception handling is required then it is recommended that all C++ compilation units be compiled with the **-Kexcept** option. Otherwise unpredictable effects may occur if an exception is thrown.

-Kexclude
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 **-Kexclude** option omits the **printf** statement from the formatted source listing if **MAX_LINE** is not currently defined with the **#define** command.
The **-Kexclude** option is equivalent to the SAS/C Compiler **exclude** option.

-Kfreg=*n*
is a synonym for the **-Gf** option.

-Kgreg=*n*
is a synonym for the **-Gv** option.

-Khlist
prints system header files in the formatted source listing. These are files that are included using the following syntax:

```
#include <filename.h>
```

The **-Khlist** option is equivalent to the SAS/C Compiler **hlist** option. See also **-Kilist**.

-Kigline
causes the compiler to ignore any **#line** statements in the input file.
The **-Kigline** option is equivalent to the SAS/C Compiler **igline** option.

-Kilist
prints user header files in the formatted source listing. These are files that are included using the following syntax:

```
#include "filename.h"
```

The **-Kilist** option is equivalent to the SAS/C Compiler **ilist** option. See also **-Khlist**.

-Kindep
generates code that can be called before the framework is initialized or code that can be used for interlanguage communication.
The **-Kindep** option is equivalent to the SAS/C Compiler **indep** option. The SAS/C Compiler and Library User's Guide, Fourth Edition covers the **indep** option in detail.

-Kjapan
translates keywords and identifiers that are in uppercase to lowercase before they are processed by the compiler. Prints messages in uppercase. This option is intended to be used with terminals or printers that support only uppercase (Roman) characters.

-Klisting[=*list-filename*]
produces a listing file for all phases of the compilation and, optionally, directs the listing to the specified file. If you do not specify a file name, the base file name of the source file will be used to construct the listing file name. See the **cool** prelinker option **-h** for information about how messages are handled when a listing is produced.

-Kloop
specifies that the global optimizer should perform loop optimizations. This option can only be used with the **-O** option.
The **-Kloop** option is equivalent to the SAS/C Compiler loop option. The **-Kloop** option and the **-Ol** option are synonyms. Refer to the SAS/C Compiler and Library User's Guide for more information about loop optimization.

Note: The behavior of the mainframe SAS/C Compiler is different from the SAS/C Cross-Platform Compiler. Loop optimization is the default on the mainframe. △

-Kmaclist
prints macro expansions. Source code lines containing macros are printed before macro expansion.
The **-Kmaclist** option is equivalent to the SAS/C Compiler **maclist** or **mlist** option.

-Knarrow
compresses the width of the listing to make it fit better on small screens.

-Knodbgcmprs
disables compression of the **.dbg370** debugging information file. By default, this information is compressed to save disk space and reduce network traffic while debugging.
The **-Knodbgcmprs** option is only meaningful when used with the **-g** option.

-Knoextname
disables the use of extended names. By default, external names that are longer than 8 characters will be accepted by the compiler, unless you specify the **-Knoextname** option. **-Knoextname** applies only to C compilations; extended names processing cannot be disabled for C++.

Note: When prelinking object modules produced by the compiler using **-Kextname**, **cool** checks for and prohibits the linking of two object modules with the same section name, by default. (See **-Ksname** option.) If **cool** detects an object module that has the same section name as a previously processed object module, it will issue an error message and exit.

Also note that you cannot use **cool** more than once on any object file that was previously compiled with the **-Kextname** option. Because **cool** resolves external references with extended names into their final form, it will not accept references that have been previously resolved. △

The **-Knoextname** option is equivalent to the SAS/C Compiler **noextname** option. For more information on extended names, refer to the SAS/C Compiler and Library User's Guide, Fourth Edition.

Note: In this release, **-Kextname** is the default. This differs from previous releases where you had to specify **-Kextname** explicitly to enable the use of extended function and identifier names. △

-Knohmulti

disables the reinclusion of system header files; these files will only be included once. (System header files are specified within angle brackets.) If **-Knohmulti** is specified, the cross-platform compiler will only include code from a header file once in a compilation. By default, the cross-platform compiler includes a copy of the header file code every time a **#include** <filename> statement is encountered, even if the file has already been included.

Note: The **-Knohmulti** option is equivalent to the SAS/C Compiler **nohmulti** option for C compilations only. △

-Knoimulti

disables the reinclusion of user header files; these files will only be included once. (User header files are specified within double quotes.) If **-Knoimulti** is specified, the cross-platform compiler will only include code from a header file once in a compilation. By default, the cross-platform compiler includes a copy of the header file code every time a **#include** "filename" statement is encountered, even if the file has already been included.

Note: The **-Knoimulti** option is equivalent to the SAS/C Compiler **noimulti** option. (Notice that the behavior of the mainframe SAS/C Compiler is different than that of the SAS/C Cross-Platform Compiler. Reinclusion of header files is disabled by default on the mainframe.) For C compilations only. △

-Knoinline

disables all inlining of functions during the optimization phase. If this option is not specified, functions specified as **inline** will be inlined by default. (Also see the **-Oin** option.)

The **-Knoinline** option is equivalent to the SAS/C Compiler **noinline** option.

-Knoinlocal

disables inlining of single-call, static (local) functions. These functions are not inlined by default during the optimization phase. (Also see the **-Oil** option.)

The **-Knoinlocal** option is equivalent to the SAS/C Compiler **noinlocal** option.

-Knolineno

disables identification of source lines in run-time messages. When **-Knolineno** is specified, module size is decreased because the generation of line number and offset tables is not required.

The **-Knolineno** option is equivalent to the SAS/C Compiler **nolineno** option.

-Knostringdup
creates a single copy of identical string constants.
The **-Knostringdup** option is equivalent to the SAS/C Compiler **nostringdup** option.

-Knousearch
specifies include-file search rules that are not typical of UNIX compilers. **-Kusearch** is the default. See “Header Files” on page 20 for additional information.

-Komd[=*omd-filename*]
invokes the object module disassembler (OMD) after successful compilation and, optionally, directs the **.omd** listing to the specified file. If you do not specify a file name, the compiler derives the listing file name from the basename of the input file with a **.omd** suffix. Also see the **-S** option.

-Koptimize
is a synonym for the **-O** option.

-Koverload
turns on recognition of the **overload** C++ keyword. If you specify this option, the translator recognizes **overload** as a reserved word; otherwise, it is treated as an identifier. For additional information, see the SAS/C Development System User’s Guide, Volume 1: Introduction, Compiler, Editor. For C++ compilations only.

-Kpagesize=*nn*
defines the number of lines per page for source and cross-reference listings. The default is 60 lines per page.

Note: The **-Kpagesize=*nn*** option is similar to the SAS/C Compiler **pagesize** option for C compilations only. △

-Kpflocal
assumes that all functions are **_ _local** unless **_ _remote** was explicitly specified in the declaration. By default, the compiler treats all function pointers as **_ _remote** unless they are explicitly declared with the **_ _local** keyword.
The **-Kpflocal** option is equivalent to the SAS/C Compiler **pflocal** option.

-Kposix
instructs the compiler to create a POSIX-compliant program by setting compile-time and run-time defaults for maximum POSIX compatibility. The **-Kposix** option has the following effects on compilation:

- The SAS/C feature test macro **_SASC_POSIX_SOURCE** is automatically defined.
- The compiler option **-Krefdef** is assumed.
- 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 **-Kposix** option, it has the following effects on the execution of the program:

- The **fopen** function assumes at run-time that all filenames are HFS filenames unless prefixed by "//".
- The **system** function assumes at run-time 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: You should not use the **-Kposix** option when compiling functions that can be used by both POSIX and non-POSIX applications. △
The **-Kposix** option is equivalent to the SAS/C Compiler **posix** option. See the SAS/C Compiler and Library User’s Guide, Fourth Edition for more information.

-Kppix
allows nonstandard use of the preprocessor.

If the **-Kppix** option is in effect, the preprocessor allows token-pasting by treating a comment in macro replacement text as having zero characters. The ANSI Standard defines the **##** 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 ANSI Standard **#** operator.

Note: The **-Kppix** option is equivalent to the SAS/C Compiler **ppix** option for C compilations only. △

-Krdepth=*n*
is a synonym for the **-Oir** option.

-Kredef
allows redefinition and stacking of **#define** names.

The **-Kredef** option is equivalent to the SAS/C Compiler **redef** option.

-Krefdef
forces the use of the strict reference-definition model for external linkage of **__rent** identifiers. The **-Krefdef** option causes the compiler to generate code that forces the use of the strict reference-definition model for reentrant external variables. If the strict reference-definition model is not used, the compiler uses the common model. This option is meaningful primarily when used with the **-Krent** or **-Krentext** options. (Strict reference-definition is always used for **__norent** identifiers.)

Because of the fact that a reference is also a definition in the common model, it is also recommended that you use the **-Krefdef** option when linking with **ar370** archives, to cause proper resolution of variable definitions.

The **-Krefdef** option is equivalent to the SAS/C Compiler and C++ Development Systems **refdef** option.

-Krent
allows reentrant modification of static and external data.

The **-Krent** option is equivalent to the SAS/C Compiler and C++ Development Systems **rent** option.

-Krentext
allows reentrant modification of external data.

The **-Krentext** option is equivalent to the SAS/C Compiler and C++ Development Systems **rentext** option.

-Krtti
enables the generation of information for RTTI on class objects that have virtual functions. By default, this option is not enabled because it increases the number of virtual function tables and the size of the information used to implement virtual function calls.

If your program uses the **dynamic_cast** or **typeid()** operators, the **-Krtti** option must be specified for each compilation unit to assure the class objects have the information required for dynamic type identification.

-Ksingleret
forces the cross-platform compiler to generate a single return sequence at the end of each function. By default, the cross-platform compiler generates a return sequence at the location of each **return** statement within a function. The main advantage of the **-Ksingleret** option is that it causes a single return from functions that have multiple **return** statements. The code to execute the single

return from the function is emitted at the end of the function, with **`return`** statements within the function causing a branch to that single return location.

`-Ksmpxivec`
generates a CSECT that is used in place of @EXTERN#. The CSECT generated by the **`-Ksmpxivec`** option has a unique name of the following form:

sname`@.`

The sname@. vector provides an alternate mechanism for reentrant initialization of **`static`** and **`extern`** data that is used with System Modification Program (SMP) update methods, which are described in Programmer's Report: SMP Packaging for SAS/C Based Products.
The **`-Ksmpxivec`** option is equivalent to the SAS/C Compiler **`smpxivec`** option. For this option to be effective, you must have the SMP libraries.

Note: The **`-Asmpxivec cool`** option must be used in conjunction with the **`-Ksmpxivec`** compiler option. The **`-Asmpxivec cool`** option builds a vector named @EXTVEC# that references the sname@. CSECT generated by the **`-Ksmpxivec`** compiler option. For example, the following command could be used to invoke the **`sascc370`** compiler driver:

```
sascc370 -Ksmpxivec -Asmpxivec filename.c
```

△
In this case, the **`-Asmpxivec`** option is passed to the prelinker. See Chapter 6, "Prelinking C and C++ Programs," on page 73 for information about the **`-Asmpxivec cool`** option.

`-Ksname=`*`sname`*
defines the section name. The **`sname`** argument can be up to seven characters in length.
The section name is assigned by the compiler using the first applicable rule in the following list:

- The section name is the name specified by the user with the **`-Ksname`** option.
- If you are using **`sasCC370`** or **`sascc370`** with the **`-cxx`** option, the section name is the first 7 characters of the basename of the input file name, neglecting any suffix.
- In the absence of a specific compile-time **`-Ksname`** option, the section name is the name of the first external function in the module, truncated to seven characters.
- If no name is provided with the **`-Ksname`** 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 provided with the **`-Ksname`** 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 **`-Ksname`** option is equivalent to the SAS/C Compiler and C++ Development Systems **`sname`** option.

`-Ksource`
outputs a formatted source listing of the program to the listing file.
The **`-Ksource`** option only controls the source listing; the cross-reference listing is requested with the **`-Kxref`** option.
The **`-Ksource`** option is similar to the SAS/C Compiler **`source`** option.

-Ksrcis=*source-filename*
specifies the name of the source file in the debugging file. This option is meaningful only when used with the **-g** option or **-Kdebug** option.

-Kstrict
enables an extra set of warning messages for questionable or nonportable code. See SAS/C Software Diagnostic Messages for more information.

Note: The **-Kstrict** option is equivalent to the SAS/C Compiler strict option for C compilations only. △

-Ktmplfunc
Controls whether a nontemplate function declaration that has the same type as a template specialization refers to the template specialization. When **-Knotmplfunc** is specified, template specializations may also be referred to by nontemplate declarations. **-Knotmplfunc** provides compatibility with older code. **-Ktmplfunc** is the default for compatibility with the ISO C++ Standard.

-Ktrigraphs
enables translation of ANSI standard trigraphs.

Note: The **-Ktrigraphs** option is equivalent to the SAS/C Compiler **trigraphs** option for C compilations only. △

-Kundef
is a synonym for the **-U** option.

-Kuse_clink
uses the program **clink** as the object code preprocessor. By default, the SAS/C C and C++ cross-compiler uses the **cool** program to prelink the object file. For more information about **clink**, see Appendix 5, "Compatibility Notes," on page 145.

-Kvstring
generates character string literals with a 2-byte length prefix. This option is used primarily in conjunction with the interlanguage communication feature.
The **-Kvstring** option is equivalent to the SAS/C Compiler **vstring** option. For more information on the **vstring** option, see the chapter about communication with other languages, in the SAS/C Compiler Interlanguage Communication Feature User's Guide.

-Kxref
produces a cross-reference listing.
The **-Kxref** option is equivalent to the SAS/C Compiler **xref** option.

-Kzapmin=*n*
specifies the minimum size of the patch area, in bytes. n refers to the number of bytes in the patch area. The default is 24 bytes.
The **-Kzapmin** option is equivalent to the SAS/C Compiler and C++ Development Systems **zapmin** option. For more information about the patch area, refer to the SAS/C Compiler and Library User's Guide .

-Kzapspace=f*n*
alters the size of the compiler-generated patch area. The size of the patch area can be increased or its generation suppressed. The default is 1.
The **-Kzapspace** 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 **-Kzapspace** 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 **`-Kzapspace`** option is equivalent to the SAS/C Compiler **`zapspace`** option. For more information about the patch area, refer to the SAS/C Compiler and Library User's Guide.

`-mrc`

causes the cross-platform compiler to generate mainframe return codes when syntax and semantic errors are detected during compilation. The mainframe return codes generated when the **`-mrc`** option is in effect are summarized in Table 3.5 on page 58.

Table 3.5 Mainframe Return Codes

Code	Definition
0	No errors or warnings found: object code is generated.
4	Warning: object code is generated and it will probably execute correctly.
8	Serious error: object code is generated but it 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 stops.

By default, the cross-platform compiler's return codes are similar to the return codes of a native UNIX compiler. In this case, a return code greater than 0 is an error. This behavior is consistent with what is expected by UNIX tools, such as **`make`**.

`-O`

executes the global optimizer phase of the compiler, which optimizes the flow of control and data through an entire function.

Global optimization includes a wide variety of optimizations, such as:

- Assigning variables to registers.
- Eliminating variable assignments that are never used.
- Moving invariant calculations out of loops.
- Replacing variables with constants whenever possible.
- Eliminating recalculation of values that have been computed previously.
- Eliminating code that is never executed.
- Changing multiplications to addition.
- Moving redundant expressions to a single, common location.

The cross-platform compiler accepts the following options to modify the operation of the global optimizer: **`-Gf`*`n`***, **`-Gv`*`n`***, **`-Oa`**, **`-Oic=`*`n`***, **`-Oid=`*`n`***, **`-Oil`**, **`Oin`**, **`-Oir=`*`n`***, and **`-Ol`**.

The **`-O`** option is equivalent to the SAS/C Compiler **`optimize`** option. See Chapter 5, "Using the Global Optimizer and the Object Module Disassembler," on page 69 for more information about the **`optimize`** option and the global optimizer. (See **`-Koptimize`**.)

`-Oa`

disables type-based aliasing assumptions. If **`-Oa`** is used, the global optimizer uses worst-case aliasing. Use of this option can significantly reduce the amount of optimization that can be performed. This option can only be used with the **`-O`** option.

The **`-Oa`** option is equivalent to the SAS/C Compiler **`alias`** option. (See **`-Kalias`**.)

`-Oic=`*`n`*
specifies the maximum complexity that a function can have and remain eligible for default inlining. The range of n is 0 to 20; with 0 specifying that only very small functions should be inlined, and 20 specifying that relatively large functions should be inlined. The **`-Oic`** option is set to 0 by default. This option is used with the **`-Oin`** option, which enables the default inlining of small **`static`** and **`extern`** functions.

The **`-Oic`** option is equivalent to the SAS/C Compiler **`complexity`** option. (See **`-Kcomplexity`**.)

`-Oid=`*`n`*
defines the maximum depth of function calls to be inlined. The range of n is 0 to 6, and the default value is 3. This option can only be used with the **`-O`** option.

The **`-Oid`** option is equivalent to the SAS/C Compiler **`depth`** option. (See **`-Kdepth`**.)

`-Oil`
inlines single-call, **`static`** functions. This option can only be used with the **`-O`** option.

The **`-Oil`** option is equivalent to the SAS/C Compiler **`inlocal`** option. (See **`-Kinlocal`**.)

`-Oin`
enables inlining of small **`static`** and **`extern`** functions, in addition to the inlining of functions defined with **`__inline`** keyword. The complexity of the functions that are inlined, other than those that are defined with the **`__inline`** keyword, is controlled by the **`-Oic`** option. The **`-Oic`** option must be specified for some non-zero n to enable the inlining of small functions. The **`-Oin`** and **`-Oic`** options can only be used with the **`-O`** option.

The **`-Oin`** option is similar to the SAS/C Compiler **`inline`** option. (Even if **`-Oin`** is not specified, functions defined with the **`__inline`** keyword will be inlined.) (See **`-Kinline`**.)

`-Oir=`*`n`*
defines the maximum level of recursive function calls to be inlined. The range of n is 0 to 6, and the default is 0. This option can only be used with the **`-O`** option.

The **`-Oir`** option is equivalent to the SAS/C Compiler **`rdepth`** option. (See **`-Krdepth`**.)

`-Ol`
specifies that the global optimizer should perform loop optimizations. This option can only be used with the **`-O`** option.

The **`-Ol`** option is equivalent to the SAS/C Compiler **`loop`** option. (Notice that the behavior of the mainframe SAS/C Compiler is different than that of the SAS/C Cross-Platform Compiler. Loop optimization is the default on the mainframe.) Refer to the SAS/C Compiler and Library User's Guide for more information about loop optimization. (See **`-Kloop`**.)

`-o` ***`filename`***
specifies the name of the output file. If the **`-c`** option is used, filename specifies the name of the output object file. Otherwise, filename specifies the name of the prelinked file. If the **`-o`** option is not specified, output is written to the **`a.out`** file by default.

The **`-o`** option is similar to the SAS/C compiler **`object`** option. Refer to the SAS/C Compiler and Library User's Guide, Fourth Edition for more information.

`-P`

creates a file containing preprocessed source code for this compilation. Preprocessed source code has all macros and **`#include`** files expanded. If the **`-P`** option is used, all syntax checking (except in preprocessor directives) is suppressed, no listing file is produced, and no object code is generated.

If **`-o`** 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 a file with a **`.i`** extension. The name of the default output file is derived from the basename of the source file.

The **`-P`** option is equivalent to the SAS/C Compiler **`pponly`** option.

`-Q`*`pathname`*

specifies an alternative pathname to be searched for the cross-platform compiler executable files. The location of the executable files that compose the SAS/C Cross-Platform Compiler (**`sascc370`**, **`lc1`**, **`lc2`**, **`cool`**, **`go`**, and **`omd`**) is defined when the SAS/C Cross-Platform Compiler is installed. This location is host specific, and is usually in the **`host`**/host-type/**`bin`** subdirectory.

`-S`

invokes the object module disassembler (OMD) after successful compilation. OMD-only options and selected compilation options are passed to the OMD, as explained in the SAS/C Compiler and Library User's Guide.

The **`-S`** option is equivalent to the SAS/C Compiler **`omd`** option. (See **`-Komd`**.)

`-Tallres`

specifies use of the all-resident library, libares.a, which is prefixed to the resident library, **`libc.a`**. The **`-Tallres`** option should be specified when developing an all-resident application. Refer to the SAS/C Compiler and Library User's Guide for more information about all-resident programs.

Note: The **`-Tallres`** option must be combined with either the **`-Tcms370`** or the **`-Tpcms370`** option to generate an all-resident application targeted for CMS. △

Note: The **`-Tspe`** option is not allowed in combination with the **`-Tallres`** option. △

`-Tcics370`

specifies that CICS is the target. The **`-Tcics370`** option causes the driver to specify the CICS libraries during linking and add the **`-m`** option to the **`cool`** command.

Note: When you use **`cool`** to link a CICS application, you will receive warnings about unresolved references to the following: **`DFHEI1`**, **`DFHEAI`**, and **`DFHAI0`**. These warning messages are expected. The output object file from **`cool`** must subsequently be moved to the target mainframe and linked with the CICS Execution Interface stubs. △

Note: If specifying the **`-Tcics370`** option causes the **`sascc370`** driver to also issue **`-m`** to **`cool`**, then using the **`sascc370`** driver option **`-Aclet`** is redundant. △

`-Tcicsvse`

specifies that CICS running under the VSE operating system is the target. The **`-Tcicsvse`** option causes the driver to specify the CICS VSE libraries during linking and add the **`-p`** option (remove pseudoregisters) and the **`-m`** option to the **`cool`** command.

Note: The **`-Tspe`** option is not allowed in combination with the **`-Tcicsvse`** option.

Note: When you use **`cool`** to link a CICS application, you will receive warnings about unresolved references to the following: **`DFHEI1`**, **`DFHEAI`**, and **`DFHAI0`**. These warning messages are expected. The output object file from **cool** must subsequently be moved to the target mainframe and linked with the CICS Execution Interface stubs. △

Note: If specifying the **-Tcics370** option causes the **sascc370** driver to also issue **-m** to **cool**, then using the **sascc370** driver option **-Aclet** is redundant. △

△

-Tcms370
specifies CMS running under VM/ESA or VM/XA as the target host operating system. The **-Tcms370** option should be specified when your application is targeted for CMS under VM/XA, VM/ESA, or VM/SP release 6.
Under VM/XA or VM/ESA, programs can run either in 24-bit addressing mode, or in 31-bit addressing mode.

Note: The cross-platform compiler generates code that is targeted for OS/390 by default. △

-Tpcms370
specifies CMS supporting System/370 mode (pre-bimodal) as the target host operating system. The **-Tpcms370** option should be specified when your CMS application will run under VM/SP release 5 or earlier. 370 mode does not support 31-bit addressing.

-Tspe
specifies use of the SAS/C SPE library, **libspe.a**, which replaces the resident library, **libc.a**. The **-Tspe** option should be specified when developing an SPE application. Refer to the SAS/C Compiler and Library User's Guide, Fourth Edition for more information about systems programming with the SAS/C Compiler.

Note: The **-Tspe** option must be combined with either the **-Tcms370** or the **-Tpcms370** option to generate an SPE application targeted for CMS.

Note: The **-Tallres** and **-Tcicsvse** options are not allowed in combination with the **-Tspe** option. △

△

-temp=*directory*
specifies an alternative directory used to store temporary files.

-U
undefines predefined macros.
Predefined macros are defined as follows:

```
#define DEBUG 1
#define NDEBUG 1
#define I370 1
#define OSVS 1
#define CMS 1
```

The definition of the DEBUG or the NDEBUG macro depends on whether you have specified the **-g** option. The **OSVS** and **CMS** macro definitions depend on the **-Tcms370** and **-Tpcms370** options. The **OSVS** macro is defined if neither the **-Tcms370** nor the **-Tpcms370** option is specified. If either of these options is specified, the **CMS** macro is defined.
The **-U** option is equivalent to the SAS/C Compiler **undef** option. (See **-Kundef**.)

-v
specifies verbose mode. In verbose mode, the command line that executes each phase of the cross-platform compiler is displayed.

-w~*n*
treats a warning condition as an error condition. The warning condition is identified by its associated message number n. Conditions whose numbers have been specified are treated as errors, and cause an error return code from the compiler. By default, a non-zero value greater than 1 is returned. If the **-mrc** option is also specified, a return code of 12 will be generated for a warning condition instead of a return code of 4.

Any number of warning conditions can be specified by entering additional **-w~** options. See also the **-w** option.

The **-w~** option is similar to the SAS/C Compiler **enforce** option.

-w+*n*
specifies that a warning, whose number is specified as n, is not to be suppressed.

Any number of warning conditions can be specified by entering additional **-w+** options. See also the **-w** option.

The **-w+** option is similar to the SAS/C Compiler **mention** option.

-w*n*
ignores a warning condition. Each warning condition is identified by its associated message number n. Conditions whose numbers have been specified are suppressed. No message is generated, and the compiler return code is unchanged. For more information about related messages, see SAS/C Software Diagnostic Messages.

Any number of warning conditions can be specified by entering additional **-w** options. If both **-w** and **-w~** specify the same message number, the warning is enforced.

The **-w** option is similar to the SAS/C Compiler **suppress** option.

The **-w** option will be passed by the driver to the appropriate phase of the compilation process, C or C++ phase. To optionally specify a compiler phase, use the syntax as described in “Specifying Phase of the Compilation of C and C++ Programs” in Chapter 3, “Compiling C and C++ Programs,” in the SAS/C Cross-Platform Compiler and C++ Development System: Usage and Reference. The format of the command is as follows:

```
-WC,-wSnnn
-Wl,-ynnn
```

where:

-WC,-wSnnn
Suppresses C++ warning and error messages, and nnn represents the message number that is to be suppressed.

-Wl,-ynnn
Suppresses C warning and error messages, and nnn represents the message number that is to be suppressed.

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 and a **`CENTRY`** macro parameter 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.

Language Extensions

This section introduces the extensions to the ISO/ANSI C language implemented in Release 6.50 of the SAS/C Compiler. Library extensions are described in SAS/C Library Reference, Volume 1, SAS/C Library Reference, Volume 2, and SAS/C Compiler and Library User's Guide.

Note: Use of these extensions is likely to render a program nonportable. △

Compiler Comment Support

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 **`-Kstrict`** compiler option is used. △

Extended @ Operator Capability

Compiler support for the **`at`** sign (**`@`**) has been extended. When the compiler option **`-KAT`** is specified, **`@`** 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.

Character and String Qualifiers

Release 6.50 introduces **A** and **E** qualifiers for character and string constants. The new qualifiers cause 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"
```

CHAPTER

4

Using the SAS/C CICS Command Preprocessor

Introduction

This chapter describes how to use the **ccp** command to request CICS services by placing CICS commands anywhere within your C or C++ source code. The **ccp** command translates the CICS commands into appropriate function calls for communication with CICS. By default, the name of the output file will be generated from the input name, with a .c suffix, unless the **-o** option is used.

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

Using the ccp Command

The SAS/C CICS Command Preprocessor can be invoked either directly or by the **sascc370** or **sasCC370** compiler driver.

Invoking ccp Directly

The following syntax is used to invoke **ccp** directly:

```
ccp [options] in_file.ccp
```

The options that may be used are described in “Option Descriptions” on page 66.

The in_file.ccp specifies the name of the input file. By default, the name of the output file will be the input filename with a **.c** suffix, unless the **-o** option is used.

Using sascc370 or sasCC370 to Invoke ccp Automatically

If your source file suffix is **.ccp**, the compiler driver uses the **ccp** command to translate any CICS commands. The **ccp** command is invoked and the output is compiled, with either the C or C++ compiler. For example, the following command could be entered to compile the file named **myfile.ccp**:

```
sascc370 -Tcics370 -Krent myfile.ccp
```

In this case, **myfile.ccp** is preprocessed for CICS commands. The resulting file is compiled with the C compiler and linked with the CICS target libraries to produce **a.out**.

```
sasCC370 -Tcicsvse -Krent myfile.ccp
```

In this case, **myfile.ccp** is preprocessed for CICS commands. The resulting file is compiled with the C++ compiler and linked with the CICS VSE library to produce **a.out**.

To pass options to the CICS preprocessor during compilation with **sascc370** or **sasCC370**, specify the compilation phase prefix, **-WP**, followed by the CICS option. For example, this command compiles **myfile.cpp** and passes the CICS **-d** option to the CICS preprocessor:

```
sascc370 -Tcics370 -WP,-d myfile.ccp
```

The options that may be used are described in the section "Option Descriptions" on page 66.

Linking CICS Preprocessed Files

You must use the **-Tcics370** or **-Tcicsvse** options on the compiler driver to cause **cool** to use the CICS or CICS VSE target libraries. However, if you are using the CICS external call interface (**-x** option), you should link with the standard resident library, **STDOBJ**, instead of the CICS libraries.

Note: When you use **cool** to link a CICS application, you will receive warnings about unresolved references to the following: **DFHEI1**, **DFHEAI**, and **DFHAI0**. These warning messages are expected. The output object file from **cool** must subsequently be moved to the target mainframe and linked with the CICS Execution Interface stubs. △

Note: For CICS, use of the **-Aclet** option is not necessary when you use **-Tcicsxxx**. However, use of one of the compiler options **-Krent** or **-Krentext** is necessary for CICS programs. See the section on compiling SAS/C programs for CICS under MVS/TSO in the SAS/C CICS User's Guide for more information. △

Option Descriptions

This section provides a description of each of the options recognized by the **ccp** command.

Note: To negate most of these options, precede the option with **!** instead of **-**. For example, to disable **-c**, use **!c**. △

-a *N,M*
adds sequence numbers to the output file. N specifies the first sequence number and M the incrementing value. If both N and M are **0**, then the output file is not

sequenced. The default for both M and N is **0**. The **-a** N,M option is equivalent to the mainframe CICS preprocessor **OUTSEQ** option.

Note: Since the cross-platform compiler does not support sequence numbers, you cannot use the **-a** option if you intend to compile the preprocessor output on UNIX. △

-b
enabled by default, causes the CICS preprocessor to produce code for the Execution Diagnostic Facility (EDF). The **-b** option is equivalent to the mainframe CICS preprocessor **DEBUG** option.

-c
enabled by default, indicates the preprocessor should translate EXEC CICS commands. The **-c** option is equivalent to the mainframe CICS preprocessor **CICS** option.

-d
causes the CICS preprocessor to process EXEC DLI commands. The **-d** option is equivalent to the mainframe CICS preprocessor **DLI** option.

-e
allows interception of all commands by the EDF. The **-e** option is equivalent to the mainframe CICS preprocessor **EDF** option.

-f ***x***
emits message only of level x and above, x may be: **I** (notes), **W** (warnings), **E** (errors) or **S** (severe errors). The default is **-f I**. The **-f *x*** option is equivalent to the mainframe CICS preprocessor **FLAG** option.

-g
shows C code generated for commands in the source listing. The **-g** option is equivalent to the mainframe CICS preprocessor **EXPAND** option.

-i
lists the CICS preprocessor options in effect on the listing file. The **-i** option is equivalent to the mainframe CICS preprocessor **OPTIONS** option.

-j
results in uppercased C keywords (**VOID**, **INT**, and so on) in the command translation. This is intended for use with the compiler's **japan** option. The **-j** option is equivalent to the mainframe CICS preprocessor **JAPAN** option.

-l ***filename***
produces a CICS preprocessor listing named filename. The **-l** filename option is equivalent to the mainframe CICS preprocessor **PRINT** option.

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

-n
handles nested comments. The **-n** option is equivalent to the mainframe CICS preprocessor COMNEST option.

-o ***filename***
writes the preprocessed output to the named file. This option cannot be negated.

-p

generates a prototype for **ccp_exec_cics** for each call to the EXEC command interface function. This is enabled by default. The **-p** option is equivalent to the mainframe CICS preprocessor PROTO option.

-r *N*

specifies the LRECL of the CICS preprocessor output file. N may be in the range 40 to 255, inclusive. The default is 255. The **-r *N*** option is equivalent to the mainframe CICS preprocessor **OUTLRECL** option.

-x

enables the CICS external call interface. This option allows a non-CICS program that is running on OS/390 to call a CICS program that is running in a CICS region. The non-CICS program can be translated, compiled, and prelinked under UNIX, but the final link-editing must take place on MVS. For example, this command prepares a program that uses the CICS external call interface for final link-editing on MVS:

```
sascc370 -WP,-x -WP,-m -Krent -Aclet myfile.cpp
```

Since the object shipped from UNIX is prelinked, you should specify the mainframe **COOL** option **NOCOOL** to run only the linkage-editor.

The **-x** option is equivalent to the mainframe CICS preprocessor **EXCI** option. For details on the CICS external call interface, see the SAS/C CICS User's Guide, Second Edition.

Note: When using the CICS external call interface, the program must be linked with the standard resident library, **STDOBJ**, and not the CICS library. △

-z

defines the number of lines per page in the listing file. The default is 60. The **-z** option is equivalent to the mainframe CICS preprocessor **PAGESIZE** option.

CHAPTER 5

Using the Global Optimizer and the Object Module Disassembler

Introduction

The global optimizer makes optimizations that improve the performance of your application in terms of execution time and program size, and the object module disassembler provides a copy of the assembler code generated for a C or C++ program. Both of these topics are covered in some detail in the SAS/C Compiler and Library User's Guide. This chapter provides a brief overview and information essential to effectively use these features with the SAS/C Cross-Platform Compiler.

The Global Optimizer

The global optimization phase optimizes the flow of control and data through an entire function. A wide variety of optimizations are performed, including the following:

- Values that are not used are eliminated.
- Calculations that do not change are moved outside loops.
- Variables whose only definition is constant are replaced by constants.
- Recalculations of previously calculated values are eliminated
- Code that can never be executed is eliminated.
- Multiplication operations within a loop are changed to addition operations.
- Expressions that are computed along all paths from a point in the code, are moved to a single, common location.

Each of these topics is treated in detail in the SAS/C Compiler and Library User's Guide.

Global Optimization Compiler Options

The **-O** compiler option is used to enable global optimization. The following options alter the behavior of the global optimizer.

Table 5.1 Global Optimizer Options

Option	**Description**
-Gfn or **-Kfreg=**n	specifies the maximum number of floating-point registers (n= 0 through 2).
-Gvn or **-Kgreg=**n	specifies the maximum number of registers the optimizer can assign to register variables (n= 0 through 6).
-Oa or **-Kalias**	specifies that the global optimizer should assume worst-case aliasing.
-Oic=n or **-Kcomplexity=**n	specifies the maximum complexity that a function can have and remain eligible for default inlining.
-Oid=n or **-Kdepth=**n	specifies the maximum depth of functions to be inlined.
-Oil or **-Kinlocal**	inlines single-call, static functions.
-Oin or **-Kinline**	enables inlining during the optimization phase.
-Oir=n or **-Krdepth=**n	specifies the maximum level of recursion to be inlined.
-Ol or **-Kloop**	specifies that loop optimizations should be performed.

Each of these options is described in Chapter 3, “Compiling C and C++ Programs,” on page 39.

Global Optimization and the Debugger

The cross-platform compiler does not optimize programs when the **-g** option is used. To use 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 **-g** option causes the compiler to suppress allocation of variables to registers, so the resulting code is not completely optimal.

You can, however, use the **-Kdbhook** option along with the **-O** option to generate optimized object code that can be used with the debugger. The **-Kdbhook** 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 **-Kdbhook** 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, which are normally used with variables, are not used when debugging optimized code. However, 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, and you can use commands such as **step**, **goto**, and **runto** to control the execution of your program. However, due to optimizations that affect register contents, the **goto** command may fail when debugging optimized code.

See Chapter 9, “Cross-Debugging,” on page 93 for more information about using the SAS/C Debugger with the SAS/C Cross-Platform Compiler.

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 or C++ program. If the object module is created with a line number-offset table (that is, if the compiler option **-l** is in effect), then the source code is merged with the assembler instructions.

Using omd

The object module disassembler can be invoked either directly or by the **sascc370** or **sasCC370** compiler driver.

Invoking omd directly

The following syntax is used to invoke **omd** directly:

```
omd [-v] object-filename source-filename
```

The object-filename argument specifies the name of a compiled object file, and the source-filename argument specifies the name of the source file used to compile the object. The **-v** option is specified to generate a verbose listing.

The output from the object module disassembler is directed to standard output when **omd** is invoked directly. A copy of the source code is merged with the disassembler listing to enable you to associate the assembler instructions with the source. If you specify the **-v** option, the listing will include a relocation dictionary, a line number-offset table, and an extended name mapping table.

Using sascc370 or sasCC370 to invoke omd

The **-S** option is used to invoke **omd** from the compiler driver. For example, the following command could be entered to compile the file named myfile.c and generate a **.omd** listing file:

```
sascc370 -S myfile.c
```

or

```
sasCC370 -S myfile.cxx
```

In these cases, the object module disassembler listing would be written to **myfile.omd**.

CHAPTER

6

Prelinking C and C++ Programs

Introduction

This chapter describes how to use the **`cool`** utility program to prelink C and C++ programs you have compiled. The first section contains a general discussion of **`cool`**. The following sections describe the syntax for invoking **`cool`** and each of the options that you can specify.

The cool Prelinker

The **`cool`** utility program is an object code preprocessor that assists in the link-editing of C and C++ programs. **`cool`** merges initialization CSECTs for static and external variables; the IBM linkage editor does not have this capability. When the **`-Krent`** or **`-Krentext`** compiler option is used to allow reentrant modification of data, the compiler creates a separate CSECT to contain the external variable initialization data for each compilation. Data to be used for the initialization of external variables are read during program start-up and copied to dynamically allocated memory. This copy process is necessary to support reentrant execution. (If no external variables are initialized in a compilation, then the CSECT is not created. When the **`-Krent`** option is used, this applies to static as well as external variables.)

If more than one compilation initializes external 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.

The **`cool`** utility 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/390 linkage editor. If any of the object files contain an initialization CSECT, **`cool`** retains the initialization data and then deletes the CSECT from the object file. When all of the object files are processed, **cool** produces a single object file containing one copy of the initialization CSECT, followed by the preprocessed object files.

The **cool** utility also checks for external variables with multiple initial values during the merge. **cool** issues a warning for external variables with multiple initial values.

When the use of extended names is specified by the **-Kextname** compiler option, **cool** performs additional preprocessing. Under the **-Kextname** option, the compiler creates special data objects in the object file that contains the original C or C++ identifiers and their associated short forms. The **cool** utility 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.

Note the following:

- You cannot use **cool** more than once on any object file that was previously compiled with the **-Kextname** option or prelinked with the **-p** option.
- When processing extended name object files, **cool** requires that each input object file have a unique section name.
- The **cool** utility also resolves external references with defining modules in **ar370** archives.

Syntax

As described in Chapter 2, "Using the SAS/C Cross-Platform Compiler and C++ Development System," on page 11, the basic syntax for invoking the **cool** prelinker is as follows:

```
cool [options] [filename1 [filename2...]]
```

The options argument can be one or more of the **cool** options listed in "Option Summary" on page 76 and described in "Option Descriptions" on page 78. Some functions requested on the mainframe using **COOL** control statements, such as **GATHER** and **INSERT**, are specified to **cool** on the workstation using options.

Note: You must use the **-o** option to specify an output file when you invoke **cool** directly. △

Specifying cool Options at Compilation

Most **cool** options have two forms:

- A form that you use when you call **cool** directly
- A form that you use when you specify **cool** options at compilation.

When you call **cool** directly, you specify one or more of the options listed in "Option Summary" on page 76. When you specify **cool** options at compilation, you specify the compiler driver form of the option. "Option Summary" on page 76 lists each **cool** option and its corresponding compiler driver form. The compiler driver form begins with **-A**. The two forms are not interchangeable: You cannot use **cool** options with the **sascc370** and **sasCC370** compiler drivers, and you cannot use the compiler driver options with **cool**.

For example, the compiler driver form of the **-p** option is **-Aprem**. The rules for specifying this option are as follows:

- When calling **cool** directly, you specify the following:

  ```
  cool -p filename
  ```

- ☐ When using the **sascc370** (or **sasCC370**) compiler driver, you specify the following:
  `sascc370 -Aprem` *filename*
- ☐ You cannot specify the following: `cool -Aprem` *filename*

Note: You can also specify **cool** options at compilation by prefixing the option with **-W**l (**-W** followed by the letter l). For details on specifying the compilation phase, refer to Chapter 3, “Compiling C and C++ Programs,” on page 39. Also see Appendix 5, “Compatibility Notes,” on page 145. △

Specifying Control Statements

Each **cool** input file may contain object code, control statements, or both. Control statements must be stored in the EBCDIC character set. Each control statement must be an 80-byte card image, padded with EBCDIC blanks if necessary. Control statements must not be separated by new-line characters. The **atoe** utility described in Chapter 8, “Conversion of Existing Programs,” on page 89 is useful when generating these card images. Any of the **cool** control statements described in the SAS/C Compiler and Library User’s Guide, Fourth Edition can be used with **cool**.

Marking and Detecting Previously Processed cool Objects

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, 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. In Release 6.50, 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**.

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 that such marking be ignored.

Prelinker Defaults

Most **cool** compiler driver options that begin with **-A** have both a positive and negative form. Most of these default to their negative form if unspecified, but a few default to their positive form.

For example, if you do not specify the **-Adupsname** option, **cool** does not allow the same SNAME to be used in multiple input files; specifying **-Anodupsname** has the same effect. Other options behave the opposite way. For example, **cool** processes extended names by default (**-Aextname**). To disable this feature, you must specify the **-Anoextname** option.

"Option Summary" on page 76 lists the default for each **cool** option with a positive and negative form.

Option Summary

The **cool** options are summarized in Table 6.1 on page 76. A more detailed description of each option is provided in the section "Option Descriptions" on page 78.

The first column lists the option's compiler driver form. This is the form of the option that you will probably use most often. The second 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. Both compiler driver options and **cool** options can be negated. To negate a compiler driver option, precede the option name with **no**. For example, to negate the **-Acontinue** option, specify **-Anocontinue**. To negate a **cool** option, precede the option name with **!** instead of **-**, for example **!p**. The third column lists the default for each option that can be negated. The fourth column lists the standalone **cool** option. This is the form of the option that you must specify when calling **cool** directly.

Note: Some options have no compiler driver form; they can be used only when calling **cool** directly. △

Table 6.1 cool Options

sascc370 Option	Negation	Default	cool Option	Description
-Aallowrecool	!	**-Anoallowrecool**	**-rc**	Specifies that the output object deck can be reprocessed by **cool**.
-Acidxref	!	**-Anocidxref**	**-xxx**	Generates an extended name **CID** cross-reference.
-Aclet (or -Acletall)	!	**-Anoclet**	**-m**	Suppresses the generation of a non-zero return code for all unresolved references, including those for extended names, and allows an output object module to be stored.
-Acletnoex	!	**-Anoclet**	**-mn**	Suppresses the generation of a non-zero return code for unresolved references to non-extended names, and allows an output object module to be stored. Unresolved references to extended names result in an error return code from cool.
-Acontinue	!	**-Anocontinue**	**-zc**	Continues processing even if a corrupted **ar370** archive is detected.

sascc370 Option	**Negation**	**Default**	**cool Option**	**Description**
-Adglib=*pn*			**-db**	Specifies a debugger file qualifier that provides for customization of the destination of the debugger file.
-Adupsname	!	**-Anodupsname**	**-zd**	Allows multiple input files to define the same SNAME.
-Aendisplaylimit=*nnn*	+		-yn*nnn*	Defines the maximum number of characters used to display extended names in messages and listings.
-Aenexit=*prog,data*	+		**-xt** *prog, data*	Invokes a user exit program with optional data.
-Agather=*prefix*	+		**-g***prefix*	Specifies a 1- to 6-character **GATHER** prefix.
-Agmap	!	**-Anogmap**	**-yg**	Includes "gathered" symbols in the listing file.
-Aignorerecool	!	**-Anoignorerecool**	**-ri**	Specifies that **cool** should ignore marks indicating it has already processed an input object deck.
-Ainsert=*symbol*	+		**-i***symbol*	Specifies an external symbol that is to be resolved by the **cool** autocall mechanism.
-Alinkidxref	!	**-Anolinkidxref**	**-xxe**	Generates an extended name **LINKID** cross-reference.
-Alist	!	**-Anolist**	**-yl**	Echo input control statements to the listing file.
-Anoextname	!	**-Aextname**	**-n**	Specifies that **cool** will not process extended names.
-Anoinceof	!	**-Ainceof**	**-zi**	Processes data after an INCLUDE statement in an input file.
-Anolineno	!	**-Alineno**	**-d**	Deletes all the line-number and offset table CSECTs.
-Anortconst	!	**-Artconst**	**-r**	Suppresses the copying of run-time constants CSECTs to the output object file.
-Anowarn	!	**-Awarn**	**-w**	Suppresses warning messages.
-Anoxfnmkeep	!	**-Axfnmkeep**	**-f**	Deletes extended function name CSECTs.
-Apagesize=*nn*	+		**-s***nn*	Defines the number of lines to print per page in the listing file.
-Aprem	!	**-Anoprem**	**-p**	Removes pseudoregisters from the output object file.

sascc370 Option	Negation	Default	cool Option	Description
-Areferences	!	**-Anoreferences**	**-xxy**	Specifies that referenced symbols and defined symbols are included in the cross-reference listing.
-Aprmap	!	**-Anoprmap**	**-yp**	Includes a pseudoregister map in the listing.
-Asmponly	!	**-Anosmponly**	**-vo**	Creates only an @EXTVEC# CSECT.
-Asmpxivec	!	**-Anosmpxivec**	**-v**	Creates an @EXTVEC# CSECT.
-Asnamexref	!	**-Anosnamexref**	**-xxs**	Generates an extended name SNAME cross-reference.
-Averbose	!	**-Anoverbose**	**-zv**	Prints additional informational messages.
-Axsymkeep	!	**-Anoxsymkeep**	**-e**	Retains extended external identifier CSECTs.
			-h[*name*]	Produces a listing and, optionally, directs the listing to the specified file.
-l*name*			**-l***name*	Identifies an **ar370** archive, **lib**name.**a**, containing files that may be included by **cool** to resolve external references.
-L*directory*			**-L***directory*	Specifies a directory that is searched for **lib**name.**a** files.
			-o *filename*	Specifies an output file.

Option Descriptions

This section provides a more detailed description of each of the options listed in "Option Summary" on page 76. The options are listed alphabetically by compiler driver name. The corresponding **cool** option is shown in parentheses. **cool** options with no compiler driver form are described last.

-Aallowrecool (**-rc** for standalone **cool**)
The **allowrecool** option 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.

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. △

-Acidxref (**-xxx** for standalone **cool**)
generates an extended name **CID** cross-reference table. The table is displayed following the other **cool** output directed to **stdout**. The extended names are displayed in alphabetical order by C identifier.
The **-Acidxref** option is similar to the mainframe **COOL** option **ENXREF(CID)**.

-Aclet or **-Acletall-m** for standalone **cool**)
suppresses the generation of a non-zero return code for all unresolved references, and allows an output object module to be stored. The **-Aclet** option is maintained for compatibility with previous releases of cool, and is equivalent to **-Acletall**.

-Acletnoex (**-mn** for standalone COOL)
suppresses the generation of a non-zero return code for unresolved references to non-extended names, and allows an output object module to be stored. Unresolved references to extended names result in an error return code from COOL when this option is specified.

-Acontinue (**-zc** for standalone **cool**)
causes **cool** to continue processing even if a corrupted **ar370** archive is detected.
The **-Acontinue** option is equivalent to the mainframe **COOL** option **CONTINUE**.

-Adbglib=*pn* (**-db** for standalone **cool**)
dbglib specifies a debugger file qualifier that provides for customization of the destination of the debugger file. On UNIX platforms, the option specified is a pathname to be prefixed to the file name. For example,

```
dbglib(/u/sasc/dbg/)
```

will generate a filename of

```
/u/sasc/dbg/sname.dbg370
```

dbglib() is the default.

Note: On UNIX platforms, the sname is capitalized and remains so for debugger filename generation. △

Note: In Release 6.50, the compiler allows the placement of the debugging information in the object file when the **dbgobj** option is specified. The **dbgobj** option is specified by default when the **-Kautoinst** option is enabled. When this information is discovered by COOL to be present in the object file, COOL will write the debugging information to a file supported by the debugger. The default filename used is somewhat different than when the debugging information is written directly by the compiler in that it is generated using the sname of the containing object. △

-Adupsname (**-zd** for standalone **cool**)
causes **cool** to permit the same SNAME to be used in more than one input file.
The **-Adupsname** option is equivalent to the mainframe **COOL** option **DUPSNAME**.

-Aendisplaylimit=*nnn* (**-yn***nnn* for stand-alone COOL)
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 display limit is set internally and cannot be overriden.
The **-Aendisplaylimit** option is equivalent to the mainframe COOL option **endisplaylimit**.

-Aenexit=*prog,data* (**-xt** *prog,data* for standalone **cool**)
invokes a user exit program with optional data. The prog argument is the UNIX pathname to the program to be invoked. The data argument is 1 to 8 characters of data to be passed to the program; data is optional. For example, you would specify the following to invoke the program at **/u/bin/myprog** and pass it the value "1 2 3":

```
-Aenexit=/u/bin/myprog,"1 2 3"
```

Note: The rules for quoting the data value are determined by your UNIX shell. △

-Agather=*prefix* (**-g** *prefix* for standalone **cool**)
causes **cool** to create data tables based on the prefix argument and append these tables to the **cool** output object code.

The **-Agather** option is similar to the GATHER control statement used with the mainframe **COOL** utility. The **-Agather** option is used primarily with C++; occasions for using the **-Agather** option are rare. Refer to the SAS/C Compiler and Library User's Guide for more information.

-Agmap (**-yg** for standalone **cool**)
causes **cool** to print a cross-reference of "gathered" symbols in the listing file.

The **-Agmap** option is similar to the mainframe **COOL** option **GMAP**. Refer to the SAS/C Compiler and Library User's Guide, Fourth Edition for more information.

-Aignorerecool (**-ri** for standalone **cool**)
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.

-Ainsert=*symbol* (**-i** *symbol* for standalone **cool**)
specifies an external symbol that is to be resolved by **cool**, if necessary. If the symbol specified by the **-Ainsert** option is not resolved after all primary input has been processed, **cool** attempts to resolve it from an **ar370** archive.

The **-Ainsert** option is similar to the INSERT control statement used with the mainframe **COOL** utility.

-Alinkidxref (**-xxe** for standalone **cool**)
generates an extended name **LINKID** cross-reference table. The table is displayed following the other **cool** output directed to the standard output device. The extended names are displayed in alphabetical order using a link id that **cool** assigns.

The **-Alinkidxref** option is similar to the mainframe **COOL** option **ENXREF(LINKID)**.

-Alist (**-yl** for standalone **cool**)
causes **cool** to echo input control statements to the listing file.

The **-Alist** option is similar to the mainframe **COOL** option **LIST**.

-Anoextname (**-n** for standalone **cool**)
specifies that **cool** will not process extended names. The **-Anoextname** option is equivalent to the mainframe **COOL** option **NOEXTNAME**.

For more information about extended names processing, see the description of the **-Kextname** compiler option in Chapter 3, "Compiling C and C++ Programs," on page 39.

-Anoinceof (**-zi** for standalone **cool**)
causes **cool** to process data following an INCLUDE statement in an input file. By default, **cool** ignores any data following an **INCLUDE** statement for compatibility with the IBM linkage editor.
The **-Anoinceof** option is equivalent to the mainframe **COOL** option **NOINCEOF**.

-Anolineno (**-d** for standalone **cool**)
deletes all the line-number and offset table CSECTs from the output object code. These CSECTs are generated by the cross-platform compiler when the **-Klineno** compiler option is used.
Line-number and offset table 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.
The **-Anolineno** option is equivalent to the mainframe **COOL** option **NOLINENO**.

-Anortconst (**-r** for standalone **cool**)
specifies that **cool** is to suppress copying the run-time constants CSECTs from the output object file. The resulting object file will be somewhat smaller but certain information used by the debugger will not be available. By default, **cool** copies these CSECTs from the input file(s) to the output file.
The **-Anortconst** option is equivalent to the mainframe **COOL** option **NORTCONST**.

-Anowarn (**-w** for standalone **cool**)
suppresses the generation of warning messages.
The **-Anowarn** option is equivalent to the mainframe **COOL** option **NOWARN**.

-Anoxfnmkeep (**-f** for standalone **cool**)
deletes extended function names CSECTs in all input object files. By default, **cool** retains these CSECTs.
The extended function names CSECT may be useful at run time 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, Third Edition for more information about 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 **-Anoxfnmkeep** option is equivalent to the mainframe **COOL** option **NOXFNMKEEP**.

-Apagesize=*nn* (**-s***nn* for standalone **cool**)
specifies the number of lines to print per page in the listing file. The default is 55 lines per page.
The **-Apagesize** option is equivalent to the mainframe **COOL** option **PAGESIZE**.

-Aprem (**-p** for standalone **cool**)
specifies that **cool** is to remove pseudoregisters from the output object file.
The **-Aprem** option is automatically enabled if either the **-Tcms370** or the **-Tpcms370** compiler option is specified.
The **-Aprem** option is equivalent to the mainframe **COOL** option **PREM**.

-Aprmap (**-yp** for standalone **cool**)
causes **cool** to include a pseudoregister map in the listing file.
The **-Aprmap** option is similar to the mainframe **COOL** option **PRMAP**.

-Areferences (**-xxy** for standalone COOL)
When the **-Areferences** option is specified along with one or more of the cross-reference options (**-Acidxref**, **-Alinkxref**, or **-Asnamexref**), referenced symbols as well as defined symbols are included in the cross-reference listing.

-Asmponly (**-vo** for standalone **cool**)
causes cool to build the @EXTVEC# vector described under the **-Asmpxivec** option. The remaining portion of the **cool** output is suppressed, so that the entire object file consists of only the @EXTVEC# CSECT.
The **-Asmponly** option is equivalent to the mainframe **COOL** option **SMPONLY**.

-Asmpxivec (**-v** for standalone **cool**)
causes **cool** to build a vector named @EXTVEC# that references the sname@ vector generated by the **-Ksmpxivec** compiler option. The sname@ vector provides an alternate mechanism for reentrant initialization of **static** and **extern** data used with SMP update methods.
The **-Asmpxivec** option is equivalent to the mainframe **COOL** option **SMPXIVEC**. The **-Ksmpxivec** cross-platform compiler option is described in Chapter 3, "Compiling C and C++ Programs," on page 39. For more information about SMP, refer to Programmer's Report: SMP Packaging for SAS/C Based Products.

-Asnamexref (**-xxs** for standalone **cool**)
generates an extended name **SNAME** cross-reference table. This table is displayed in alphabetical order, sorted on the **SNAME** associated with each object file, following the other **cool** output directed to **stdout**.
The **-Asnamexref** option is similar to the mainframe **COOL** option **ENXREF(SNAME)**.

-Averbose (**-zv** for standalone **cool**)
causes **cool** to produce extra messages about its processing. These messages are displayed on the standard error device, and are included in the listing if a listing file is being produced. These messages are particularly useful for determining how symbols are resolved.
The **-Averbose** option is equivalent to the mainframe **COOL** option **VERBOSE**.

-Axsymkeep (**-e** for standalone **cool**)
specifies that the extended external identifier CSECTs in all input files are retained. By default, these CSECTs are not retained.

Note: Retaining the extended function names CSECT or the extended external identifier CSECT makes the resulting prelinked object file somewhat larger. △
The **-Axsymkeep** option is equivalent to the mainframe **COOL** option **XSYMKEEP**.

-h [*name*] (standalone **cool** only)
produces a listing and, optionally, directs the listing to the specified file. If you do not specify a file name, the listing is directed to **stdout** (the standard output device). The listing file contains a list of the options that are in effect and copies of any diagnostic messages. All **cool** messages are directed to the listing file and to **stderr** (standard error device). "Trivial" messages, like banners and the **cool** return code message, are directed to the listing file and also to **stdout**, if **stdout** is not the listing and the **-zv** (verbose) option is specified.
When you specify any of the following **cool** options, **-h** is assumed: **-yl**, **-yp**, **-yg**, **-xxe**, **-xxs**, **-xxx**.
You can also use the **-Klisting** compiler option to produce a listing for all phases of the compilation, including the prelinking phase. Message handling is the same as **-h**. When you specify any of the following **cool** options during compilation, **-Klisting** is assumed: **-Alist**, **-Aprmap**, **-Agmap**, **-Alinkidxref**, **-Asnamexref**, **-Acidxref**.

-l*name*
identifies an **ar370** archive containing members that may be included by **cool** to resolve unresolved external references. The name parameter specifies the filename of the **ar370** archive. **cool** will look for the archive named **lib**name**.a**

Note: There must not be a space between **-l** and name. △

The **-l** option has no effect unless the **-L**directory option is also specified.

The **-l** option is similar to the mainframe **COOL** option **ARLIB**.

-L *directory*

specifies the name of a directory to be searched for **ar370** archives.

Note: There must not be a space between **-L** and directory. △

The **-L** option is similar to the ARLIBRARY control statement used with the mainframe **COOL** utility.

-o*filename* (stand-alone **cool** only)

names the file in which the **cool** utility stores its output. This option must be specified when **cool** is invoked as a stand-alone utility. If **cool** is invoked by **sascc370** or **sasCC370**, the output is directed to the **a.out** file by default unless the **-o** compiler option is specified.

CHAPTER 7

ar370 Archive Utility

Introduction

The **ar370** archive utility is used to maintain groups of files that are combined into a single archive file. Normally, the **ar370** archive utility is used to generate an archive containing object files used by **cool** to resolve external references.

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. For each object file 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 required between an **ar370** member name and the external symbol names defined by the member.

Physically, each **ar370** archive is composed of three parts.

Table 7.1 ar370 Components

Component	Contents
header	information such as the date of the last modification and the release number of the **ar370** utility that made the modification.
member archive	a copy of each file added to the library. (For **ar370** archives, unlike the files in a UNIX directory, the order of members may be significant.)
symbol table	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.

CAUTION:
ar370 archives are created and maintained only by **ar370**.
The internal structures and the data they contain are in EBCDIC. **ar370** archives should never be modified or accessed in any way, other than through **ar370**. △

Note: The SAS/C Cross-Platform Compiler includes two utilities that are useful for working with **ar370** archives: **ar2updte** and **updte2ar**. For details, see Appendix 3, "ar2updte and updte2ar Utilities," on page 127. △

Using the ar370 Utility

The **ar370** archive utility is invoked directly with the following command:

```
ar370 cmds [posname] libname [fname...]
```

The cmds argument must be specified, and consists of an optional -, followed by one of the command characters **d**, **m**, **r**, **x**, or **t**. (**t** may be specified with any other command.) Optionally, you can concatenate the command character with one or more of the command modifier characters **a**, **b**, **e**, **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 libname argument specifies the filename of the archive library and must be present. By convention, an extension of **.a** is used to identify archive library files.

Each fname argument specifies the name of a file to be added or replaced, or the name of an archive member to be manipulated. Member names must be specified exactly as they appear in the archive.

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.txt** and **walk.txt** in the **ar370** archive named **zoom.a**. Verbose output is requested.

```
ar370 rv zoom.a run.txt walk.txt
```

If either **run.txt** or **walk.txt** does not exist in the **zoom.a** archive, it is added to the archive by the **r** command character.

When performing an add or replace, the name of the archive member is generally derived from the file name. You can specify an alternate name for the archive member by following the complete input file name with the replace-as operator '=' and the alternate name for the archive member. For example:

```
ar370 r mylib this.text.a1=that.obj
```

This command stores the file **this.text.a1** in the archive **mylib** with the member name **that.obj**.

Command Characters

The following command characters are recognized:

Table 7.2

Command Character	Description
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, specifying 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) the posname member. 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 about only those members appears. 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:

Table 7.3

Modifier Characters	Description
a	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	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 symbols (defined and referenced) in the specified members are displayed.

Modifier Characters	Description
j	Japan or uppercase: Produces all 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, we recommend that you back up the archive so that you will not lose your data in the event that the original dataset is destroyed. *Note:* The **q** command modifier 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 **t**, 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** modifiers, the **v** modifier causes the **ar370** archive utility to print each command modifier character and the member name associated with that operation. For the **r** modifier, 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.

Combinations of Commands and Modifiers

Only the combinations of commands and command modifiers shown in the following table are meaningful.

Table 7.4 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**
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**

CHAPTER
8

Conversion of Existing Programs

Introduction

In an effort to take full advantage of the workstation environment, the SAS/C C and C++ cross-platform compilers have been designed to be very similar to a UNIX C compiler. Because of this, some changes may be needed when you first move your application from a native compilation environment to a cross-development environment. These suggested changes do not alter your application's ability to be built in the native environment.

Source Code Changes

The SAS/C C and C++ cross-platform compilers support the same set of digraphs supported by the SAS/C Compiler on the mainframe. However, to aid in portability, it is advisable to replace these digraphs with characters that are readily available on the host workstation. The **`mf2unix`** and **`unix2mf`** utilities described later in this chapter can assist with this conversion.

Also, when moving source code from the mainframe to the workstation, any sequence numbers must be removed, since the SAS/C C and C++ cross-platform compilers do not support sequence numbers.

Filename Changes

The SAS/C C and C++ cross-platform compilers do not translate filenames found in **`#include`** statements. These filenames are case-sensitive and no truncation occurs. Unlike the mainframe, no special consideration is given to case in include-file processing; therefore, source code that contains uppercase filenames in **`#include`** statements will probably require alteration.

Utility Programs

The cross-platform compiler provides the following utilities to help you port applications from the mainframe to the workstation:

Table 8.1 Utility Programs

Utility	Description
`mf2unix`	translates source code from mainframe format to UNIX format.
`unix2mf`	translates source code from UNIX format to mainframe format.
`etoa`	translates text from EBCDIC to ASCII.
`atoe`	translates text from ASCII to EBCDIC.
`objdump`	prints out a mainframe object file for viewing on the host workstation.

Each of these utility programs is described in the following sections.

Note: If your object code is currently stored in a PDS on the mainframe, you may also find the **`updte2ar`** utility useful. For details, see Appendix 3, "ar2updte and updte2ar Utilities," on page 127. △

mf2unix and unix2mf

The **`mf2unix`** utility program translates source code from mainframe to UNIX format, and the **`unix2mf`** utility program is used to translate source code when porting in the opposite direction, that is, from UNIX to mainframe format. The syntax for invoking these utilities is as follows:

```
mf2unix [option...]
```

```
unix2mf [option...]
```

Both utilities take source code from standard input, perform the translation, and then print the output to the standard output. The translations performed by the two utilities are different:

- **`mf2unix`** strips off trailing blanks and sequence numbers, and changes digraphs to brackets.
- **`unix2mf`** breaks long lines, changes brackets to digraphs, and replaces tabs with the correct number of blank space characters.

mf2unix Options

The following option arguments can be specified to modify the translation performed by the **`mf2unix`** utility:

Table 8.2 mf2unix Options

Option	Description
-lbl	leave trailing blanks.
-offdi	do not change digraphs to brackets.
-recfm format-type	specifies the record format of the input file. format-type can be either a **v** to indicate variable-length records, or **f** to indicate fixed-length records.
-noseq	do not remove sequence numbers.

unix2mf Options

The following option arguments can be specified to modify the translation performed by the **unix2mf** utility:

Table 8.3 unix2mf Options

Option	Description
-l num	defines the output line length, where num is the maximum number of characters in a line. The default line length is 72 characters.
-t num	specifies the number, num, of blank space characters used in place of the tab character. If num is not specified, the default is 8.
-offt	do not replace tabs with blank space characters.
-offdi	do not replace brackets with digraphs.

Examples

In the following example, the **mf2unix** utility is used to translate the source code contained in **native.c** from mainframe to UNIX format, redirecting the output to **cross.c**:

```
mf2unix -lbl < native.c > cross.c
```

The **-lbl** option specifies that trailing blanks should not be removed. (Notice that the < and > redirection operators are used to redirect the input and output of **mf2unix**.)

In the next example, the **unix2mf** utility translates the source code contained in the file **cross.c** from UNIX format to mainframe format:

```
unix2mf -t 10 < cross.c > native.c
```

Output is redirected to **native.c** and the **-t 10** option specifies that tabs should be replaced with 10 blank space characters instead of the default of 8.

etoa and atoe

The **etoa** utility performs an EBCDIC-to-ASCII translation, and the **atoe** utility performs an ASCII-to-EBCDIC translation. Both utilities read from standard input, perform the translation, and then write to standard output. No assumptions about input file format are made, with regard to new-lines or any other record format. The utilities simply copy the bytes while performing the translation. Also, both utilities use the same translation tables used by the SAS/C C and C++ cross-platform compilers. (IBM code page 1047 standard).

Examples

The **etoa** and **atoe** utilities do not accept input or output filename arguments; therefore, the most effective way to use these utilities is to redirect the input and output files. For example, the following command redirects the input from native, performs an EBCDIC-to-ASCII translation, and then redirects the output to cross:

```
etoa < native > cross
```

Another way to effectively control input is with a pipe. For example, the output from the operating system's **cat** command can be piped to the **atoe** utility as follows:

```
cat native | atoe > cross
```

In this example, the input file, **native**, is copied to the standard output file, **cross**, with ASCII-to-EBCDIC translation performed by **atoe**.

objdump

The **objdump** utility prints out a mainframe object file (either OS/390 or CMS) for viewing on a host workstation. Output from the utility is directed to the standard output file and is printed in 80-column lines, with EBCDIC characters translated to ASCII. (**objdump** uses the same translation tables as used by **etoa** and **atoe**.) The resulting output is similar to what you would see if you were to browse the file using the ISPF editor under MVS.

The syntax used to invoke **objdump** is as follows:

```
objdump [option...] object-file
```

The input file is specified by the object-file argument, and the option arguments can be either of the following:

Table 8.4 objdump option Arguments

Option	**Description**
-e	Specifies that no EBCDIC-to-ASCII translation is to be performed. The output will remain in EBCDIC characters.
-h	specifies HEX ON. Each line of the output is followed by two lines representing the hexadecimal values of the bytes; as if the HEX ON command had been given in the ISPF editor.
-n	specifies numbered output. Each 80-column line of output is preceded by a line number.

Example

Assuming that **foo.o** is the output of the cross-platform compiler, the following command directs a dump of that object file to **stdout**.

```
objdump -h -n foo.o
```

In this case, both hexadecimal values and line numbers are displayed in the output.

CHAPTER 9 Cross-Debugging

Introduction

The SAS/C Debugger provides the capability of debugging programs in a cross-development environment. To debug a load module that was compiled with the SAS/C or C++ cross-platform compiler, you simply run the program with the **`=debug`** runtime option, just like any other SAS/C or C++ load module.

The debugger provides access to information from several different types of files, which may be resident on either the UNIX host or the target mainframe, including:

- System Include Files
- User Include Files
- Source Files
- Alternate Source Files
- Debugger Files

When developing an application in a cross-development environment, the files used by the debugger, with the exception of the load module, may reside on the host workstation. In order for the debugger to access files that reside on the workstation, a distributed file system must be used to establish a client/server relationship between the target mainframe and the host workstation. The distributed file system used in the SAS/C cross-development environment is the Network File System (NFS) described in Appendix 1, “Installing and Administering the NFS Client,” on page 107 and SAS Technical Report C-113 SAS/C Connectivity Support Library, Release 1.00. Using NFS, the debugger, running on the mainframe under OS/390 or CMS, has direct access to the source, include, and debugger files that reside on the host workstation.

If the debugger’s default file searching mechanism does not meet your needs, you can change or augment the search mechanism with the debugger’s **`set search`** command.

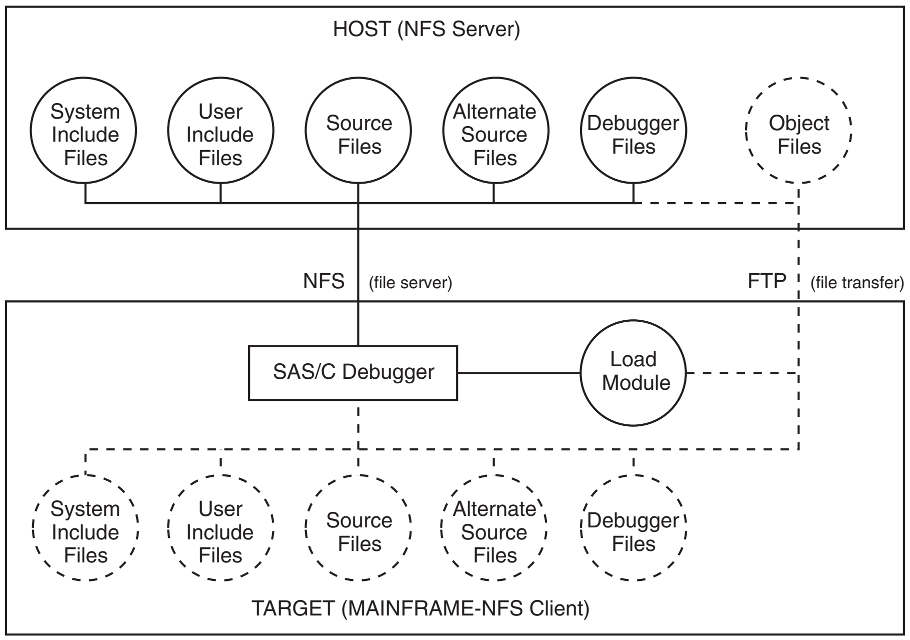
The **`set search`** command is used to specify filename templates. Filename templates are used to specify the identity and location of the source, include, or debugger files associated with the load module being debugged. Multiple filename templates can be defined for each type of file. As a result, the debugger can search for a file by more than one name or in multiple locations. Each template is saved in a search list, and each search list is associated with a specific type of file.

Filename templates are character strings which are similar to the patterns used in a C **`printf`** statement. Each filename template may contain conversion specifiers and characters. A conversion specifier is a character or a string preceded by a percent character. The conversion specifier is either replaced by its associated string or specifies the format of the conversion specifier that follows it. The resulting string is used as the name of the file to be opened. If a file with the resulting name cannot be opened, the next filename template in the search list is processed until either a file is opened or there are no more filename templates in the search list for that type of file.

This is a very powerful technique that allows you to direct the debugger to files that have moved or even changed names or file systems. This chapter explains how to use the **`set search`** and **`set cache`** commands to define filename templates and establish search lists.

Figure 9.1 on page 94 illustrates the relationship between the files used by the SAS/C Debugger in the cross-development environment.

Figure 9.1 Debugging in a Cross-Development Environment



Using the SAS/C Debugger in a Cross-Development Environment

To debug a program in the cross-development environment, you should perform the following steps:

1. Compile the program on the host workstation, using the `-g` option to specify generation of a debugger file.
2. Create a load module for your program that resides on the target mainframe.
3. Use the NFSLOGIN command to access the NFS server network from the mainframe. See "Logging on to the NFS Network" on page 118 for more information.
4. Mount the workstation's file system from your mainframe client using one of the methods described in "Accessing Remote File Systems" on page 119.
5. Invoke the debugger, using `set search` commands in the debugger PROFILE to specify search lists for the source, include, and debugger files.

   Note: The debugger uses standard fopen calls to access these files. If you encounter difficulty accessing files, the problem may be caused by your remote file mount, and the failure to properly match the mount point and the templates in the debugger's search lists. △

If you do not use the `set search` command to specify search lists, the debugger resorts to its default search mechanism, using the filenames contained in the object and debugger files to locate files. By default, the debugger uses the `path`: filename style prefix with workstation filenames. The `path`: prefix is described in Appendix 1.

The next section explains how to use the debugger's `set` command to specify search lists and a cache location for the debugger file. You should refer to the SAS/C Debugger User's Guide and Reference for additional information about the SAS/C Debugger.

Using the Debugger's set Command

The SAS/C Debugger's `set` command provides two subcommands: `set search` and `set cache`. The `set search` command is used to specify a search list consisting of one or more filename templates. Each filename template specifies a location used by the debugger to search for source, include, or debugger files associated with the load module being debugged. The debugger traverses the search list, looking for the file specified by each filename template.

The `set cache` command is used in cross-development environments that support a distributed file system, primarily to improve the debuggers performance when accessing debugger files. The benefit is especially noticeable when debugger files are large. This command uses a filename template to specify the primary location to save and search for debugger files. In a typical cross-debugging session, this location would be on the mainframe.

Note: Frequently, file access problems are caused by an improper mount to the remote file system. If you encounter difficulty with either the `set search` or the `set cache` subcommands, refer to "Accessing Remote File Systems" on page 119. △

Locating the Debugger File

Load modules that have been generated from objects compiled by the SAS/C compiler contain filename information for the debugger file. The format of this filename information depends on the host that performed the compilation and the file system the debugger file was created in. When debugging a program compiled by the SAS/C Cross-Platform Compiler, the debugger will look for the debugger file in the following locations in the order listed:

1 Any cache location, as specified by the **`set cache`** command.
2 Any locations in the debug search list, as specified by the **`set search debug`** command.
3 The original file name the compiler used to open the file when it was created.
4 The file name the compiler used to open the file when it was created with the SAS/C filename style prefix **`path:`**.

The debugger first checks to see if a cache location has been specified. The **`set cache`** command uses a filename template to specify a location for the debugger file. For example, the following form of the **`set cache`** command could be used to specify a cache location in the CMS file system:

```
'SET CACHE DEBUG = "%sname dbg370"'
```

If the debugger file is found in the cache location, that file is opened. If the debugger file is not found in the cache location or the module has been recompiled since the debugger file in the cache location was last copied, the debugger continues to search for the file by performing the remaining steps in the search order. If the debugger file is found, it is then copied to the specified cache location and the new cache file is used.

If no cache location was specified or a debugger file is not found in the cache location, the debugger will attempt to find the debugger file using any filename templates defined in the debug search list. On OS/390 systems, the debugger has a default search list for debugger files which, is equivalent to the command:

```
set search debug = "//ddn:DBGLIB(%sname)"
```

Note: You can create an empty debug search list with a **`set search debug`** command of the form: **`set search debug =`** `""`. △

On CMS systems, no default templates are defined for the debug search list, so you will probably want to define one or more templates. The following form of the **`set search`** command can be used to specify a new search list for the debugger file:

```
'SET SEARCH DEBUG = "cms: %sname db *"'
```

If the debugger file is not found using the debug search list, then the debugger will attempt to open the file by the name the compiler used when it created the file.

Finally, the debugger will attempt to open a file with the name the compiler used when it created the file and the SAS/C filename style prefix **`path:`**.

Locating Source Files

The debugger file contains filename information for the source and alternate source files used to compile your program. The debugger will look for the source file in the following locations in the order listed:

1 Any locations in the source search list, as specified by the **`set search source`** command.
2 The original file name the compiler used to open the file when it was created.
3 The file name the compiler used to open the file when it was created with the SAS/C filename style prefix **`path:`**.

On OS/390 systems, the debugger uses a default search list for source files, which is equivalent to the following command:

```
set search source = "//ddn:DBGSRC(%sname)"
```

If a file is not found using one of the templates in the source search list the debugger attempts to open the file by the name the compiler used for the file. Finally, the debugger will attempt to open a file with the name the compiler used when it created the file, prefixed by the SAS/C filename style prefix **`path:`**.

The source search list is not checked for source files that have been altered by a **`#line`** preprocessor statement that specified a filename. Instead, the separate **`altsource`** search list is used. See Table 9.2 on page 99 for more information on **`altsource`**.

You can use the following forms of the **`set search`** command to specify a new source search list to be used to locate these files:

```
set search source = "template1" "template2"...

set search altsource = "template1" "template2"...
```

Locating include Files

The debugger file also contains filename information for the system include and user include files used to compile your program. The different types of include files each have a separate search list. The debugger will look for an include file in the following locations in the order listed:

1 Any locations in the associated search list, as specified by the **`set search systeminclude`** command or the **`set search userinclude`** command.

2 The original file name the compiler used to open the file when it was created.

3 The file name the compiler used to open the file when it was created with the SAS/C filename style prefix **`path:`**.

You can use the following forms of the **`set search`** command to specify a new search list to be used to locate these files:

```
set search systeminclude =
   "template1" "template2"...

set search userinclude =
   "template1" "template2"...
```

Debugger Performance Considerations

A distributed file system makes it possible to develop your applications in a cross-development environment. In a distributed file system, programs can read or write files directly in a file system on a remote machine. The Network File System (NFS) client support provided by the SAS/C Connectivity Support Library allows the SAS/C Debugger to access files that do not reside on the mainframe at all. Additional information can be found in Appendix 2, "Using the NFS Client," on page 117.

The main performance issue to consider when debugging in a cross-development environment is the time required by the debugger, which runs on the mainframe, to access files residing on the host workstation. In general, if you can reduce the number of times files that reside on the workstation are accessed by the debugger, performance will be improved.

One method of improving debugger performance is to use the **`set search`** command to direct the debugger to access files residing on the mainframe whenever possible. For example, when developing in a cross-development environment, it is likely that identical copies of the system include files will reside on both the host workstation and the target mainframe. You should use the **`set search systeminclude`** command to direct the debugger to use the system include files located on the target mainframe.

Another way to improve performance is to specify a debugger Source Window buffer that is large enough to hold the entire source file. This allows the debugger to keep the entire source file in mainframe memory for the time that the compilation is being debugged. Switching compilations causes the file to be flushed. As a guideline, the amount of memory needed to hold one source line is equal to the length of the line, after stripping trailing blanks, plus three bytes. Refer to documentation for the Config Window and the **`window memory`** command in the SAS/C Debugger User's Guide and Reference for more information about debugger window buffers.

Even though your source, include, and debugger files may reside on the host workstation, on systems that do not enjoy the advantages of a distributed file system, or if your situation requires you to minimize network traffic, it may be advantageous to use a file transfer mechanism, such as FTP, to copy some of these files to the target mainframe. For example, if you are debugging an application composed of many source files and you are only actively developing the code in one or two of those files, the performance of the debugger will be improved if the source files that will not require frequent changes and re-compilation reside on the target mainframe as well as the host workstation.

Similarly, you may use the **`set cache`** command to establish a cache location for your debugger file if you feel this appropriate for the application being debugged.

set Command Reference

The SAS/C Debugger's **`set`** command is best used in the debugger PROFILE to specify search lists for source, include, and debugger files, as well as a cache location for your debugger file. However, the **`set`** command may also be issued on the command line. The following reference section describes both the **`set search`** subcommand and the **`set cache`** subcommand.

set

Controls file access.

ABBREVIATION

```
se{t}
```

FORMAT

```
set subcommand subcommand-arguments
```

DESCRIPTION

The **`set`** command has two subcommands: **`search`** and **`cache`**. The **`set search`** command is used to control the search templates that are used to access debugger and source files, and the **`set cache`** command is used to specify a cache location for debugger files. The **`set cache`** command also uses a template to specify this location.

The **`set search`** and **`set cache`** subcommands are described in the following paragraphs.

search SUBCOMMAND

The **`search`** subcommand is used to establish a search list, control tracing, add, or remove templates from a search list. The **`search`** subcommand has the following forms:

Table 9.1 search Subcommand Formats

Format	Example
1	**`set search`** file-tag **`=|+|-`** “template1" ["template2” ...]
2	**`set search`** file-tag **`=`**
3	**`set search`** file-tag **`|* ?`**
4	**`set search`** file-tag **`|* trace on|trace off`**

The file-tag argument specifies the type of file that a template applies to and can be any of the following:

Table 9.2 file-tag Values

Type of file	Description
`debug`	specifies that the template is for debugger files.
`source`	specifies that the template is for source files.
`altsource`	specifies that the template is for alternate source files. (An alternate source file refers to source code altered by a **`#line`** preprocessor statement that specifies a filename.)
`systeminclude`	specifies that the template is for system include files.
`userinclude`	specifies that the template is for user include files.

Format 1: This format of the **`set`** command specifies a search list for the type of files designated by file-tag. Each search list consists of one or more templates that are used by the debugger to locate debugger or source file types.

The =|+|- argument is used as follows:

Table 9.3 set command Operations

Argument	Description
=	sets the search list equal to the specified templates.
+	appends the specified templates to the search list.
-	removes all occurrences of the specified templates from the search list.

The template arguments define the search list. Each template argument uses one or more of the following conversion specifiers to define a template used by the debugger to generate filenames:

Table 9.4 template Arguments

Value	Description
%lower or **%l**	causes the replacement text for the conversion specifier following the **%lower** to be converted to lowercase. The character after the **%lower** or **%l** must be the start of another conversion specifier.
%upper or **%u**	causes the replacement text for the conversion specifier following the **%upper** to be converted to uppercase. The character after the **%upper** or **%u** must be the start of another conversion specifier.
%sname or **%s**	is replaced by the section name of the program being debugged. (The section name must have been specified when the program was compiled.) The section name is always uppercase, if a lowercase version is required, prefix the **%sname** or **%s** specification with **%lower**.
%fullname	is replaced by the entire filename stored in the object or debugger files. The format of the filename is implementation dependent and this conversion specifier should not be used unless you have complete knowledge of the filename stored in the object or debugger files. This conversion specifier is most useful for alternate source files, where it will be replaced by the complete filename that appears in the **#line** statement.
%leafname or **%lf**	is replaced by the portion of the filename stored in the object or debugger files after the last slash, if present. If there is no slash, it is the entire filename stored in the object or debugger files.
%basename or **%b**	is replaced by the portion of **%leafname** that is before the last dot. If there is not a dot in **%leafname**, then **%basename** is the same as **%leafname**.
%extension or **%e**	is replaced by the portion of **%leafname** that is after the last dot. If there is not a dot in **%leafname**, then **%extension** is set to a null string.
%m	is replaced by the member name of the original source file if it was a member of PDS.

You can include a percent character (**%**) in a template by specifying two percent characters successively (**%%**).

The filenames generated by the application of the conversion specifiers in the template are passed to the **fopen** function, which opens the appropriate file for the debugger to access. If these files are located on a remote host, the SAS/C Connectivity Support Library is used to establish an NFS connection between the local and remote host.

For example, to use SAS/C Connectivity Support Library to access files on a UNIX workstation, the following template could be specified:

```
"path:dbgfiledir/%leafname"
```

If **`%leafname`** consists of a base and an extension, a functionally equivalent template could be specified as follows:

```
"path:dbgfiledir/%basename.%extension"
```

A similar template could be specified to access files on MVS. For example, the following template would access a PDS member that matches **`%basename`**:

```
"dsn:userid.proj4.h(%basename)"
```

Format 2: The second form of the **`set search`** command is used to remove all of the search templates associated with a file-tag. It specifies a null search list.

Format 3: The question mark (**`?`**) character is used to display the search list associated with a file-tag. An asterisk (`*`) can be used as a wildcard character in place of a specific file-tag argument. Specifying **`set search * ?`** will display the search lists for all debugger and source files, including the cache location, if it was specified with a **`set cache`** command.

Format 4: The final form of the **`set search`** subcommand is used to turn tracing on or off. When tracing is turned on, the debugger displays a message each time it attempts to open a file, possibly using a filename generated by a template. The message displays the name of the file the debugger was looking for and whether or not the search was successful.

An asterisk (`*`) can be used as a wildcard character in place of a specific file-tag argument. If an asterisk is specified for the file-tag, tracing will be affected (either turned on or turned off) for debug, source, altsource, systeminclude, and userinclude files.

Reattempting a set search

Release 6.50 of the SAS/C Debugger enhances the **`set search`** capability introduced in Release 6.00. In Release 6.00, only one attempt to locate the debugger or source files would be made. This meant that if the **`set search`** commands located in the profile were not correct, or had not been corrected before an attempt to load the source, the current debugging session would have to continue without access to that particular source.

In Release 6.50, if a **`set search`** command is issued, followed by a **`list`** command, the debugger attempts to load any files that were not previously found, using the modified **`set search`** templates. For example, if an attempt to load a source file fails because the source files have been moved to the dataset SASC.APPL.SOURCE, issuing the command :

```
set search source+"dsn:sasc.appl.source(%basename)"
```

followed by a **`list`** command causes the debugger to reattempt the search for the source.

The **`set search`** issued does not have to directly correlate to the failed search. For example, a common problem encountered when debugging, is to forget to allocate the DBGLIB dataset definition. When the debugger fails to locate the debugger file, a command such as:

```
system alloc fi(dglib) dsn(appl.dbglib)shr
```

could be issued to allocate the DD. A ‘dummy’ **set search** command could then be issued. For example:

```
set search altsource+""
```

followed by a **list** command will cause the search to be reattempted.

cache SUBCOMMAND

The **set cache** command is used to specify a cache location for the debugger file. (In a cross-development environment, the original debugger file may be located on the host workstation and the cache location will be on the target mainframe.) A cache location is specified to provide faster access to debugging information.

The format for the **set cache** subcommand is as follows:

Format: **set cache debug =** "template"

Notice that **debug** is the only valid type of file for the **set cache** subcommand.

The template argument is described in the previous section and is used to specify the cache location. When debugging a program, the debugger first looks for the debugger file in the cache location. If the debugger finds a current version of the debugger file in the cache location, then the debugger uses the file. If a debugger file is not found in the cache location, or if the debugger file in the cache location is not current, then the current debugger file is copied to the cache location. However, if the cache file is not a valid debugger file, it will not be overwritten by the debugger.

EXAMPLES

```
set search userinclude =
   "path:/usr/c/headers/%leafname"
```

specifies a search list for user include files. When the debugger looks for source code that was included from a user include file located on a host workstation, this template is used to generate a filename and open the file on the workstation.

```
set search source =
   "hfs:/home/cxx/src/%leafname"
```

specifies a search list for source files in the OS/390 UNIX System Services hierarchical file system (HFS). The **hfs:** filename style prefix instructs the debugger to look for the file in the HFS file system and open the file if it is found.

```
set search userinclude +
   "dsn:userid.c.headers(%basename)"
```

specifies a template that is appended to the search list for user include files that was established in the previous example. This template generates an OS/390 **dsn:** style filename that is searched if the user include file is not found on the workstation.

```
set search userinclude trace on
```

turns tracing on for user include files. Whenever the debugger searches for a user include file, a message will be displayed telling you the name of the file searched for and if the search was successful or not.

```
set search userinclude ?
```

displays the search template list used to generate filenames for user include file searches.

```
set search userinclude =
```

resets the search template list for user include files to null.

```
set cache debug = "dsn:userid.cache.db(%sname)"
```

specifies an OS/390 data set used to cache the debugger file on the target mainframe.

```
set cache debug = "cms:%sname dbg370"
```

specifies the location of a CMS file used to cache the debugger file on a target mainframe.

SYSTEM DEPENDENCIES

The filenames generated by the search templates are dependent upon the names the compiler used to open the files originally, which are operating system dependent.

COMMAND CAN BE ISSUED FROM

debugger start-up file	**yes**
command line	yes
configuration file	no
Source window prefix	none

SCOPE

The **set** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0
Unsuccessful: 1

P A R T 2

Appendices

APPENDIX

1

Installing and Administering the NFS Client

Introduction

In a cross-development environment, the Network File System (NFS) client support provided by the SAS/C Connectivity Support Library (CSL) enables the SAS/C Debugger to communicate with the host workstation. This appendix provides the basic information necessary to administer this NFS support. Additional information is contained in Appendix 2, “Using the NFS Client,” on page 117 and SAS Technical Report C-113, SAS/C Connectivity Support Library, Release 1.00.

As an administrator for the SAS/C CSL NFS client, you must be concerned with installing the software, establishing access controls for remote file security, (optionally) developing file-system mount configurations, and diagnosing problems. This support is provided in a distributed file systems environment that uses Sun NFS protocol for network communication between computer systems.

Distributed File Systems

As networking protocols and applications have become more sophisticated, file sharing among computers has evolved from simple file transfer to the construction of distributed file systems. In a distributed file system, programs and users can access (open, read, write, etc.) file systems from a remote machine directly, as if they were attached to the local system.

Although numerous designs for distributed file systems have been implemented experimentally, only a few have achieved commercial success. Of these, the Sun Microsystems Network File System (NFS) protocol is by far the most widely used. Although not as full-featured as some other file systems (most notably the Andrew File System) in areas such as file caching and integrated security administration, its simple and modest design have made it easy to implement on a wide variety of systems. NFS software is currently available for almost every computer and operating system on the market today.

NFS Design

NFS is implemented using a protocol composed of Sun Remote Procedure Call (RPC) function calls. As with most RPC applications, the protocol supports a dialog among servers and clients. The NFS servers are the machines that provide remote access to their file systems. NFS clients are programs that access the files on another system. Use of RPC enhances interoperability among diverse machines.

The NFS protocol views all file systems as conforming to the hierarchical directory organization that has been popularized by the UNIX operating system and that was subsequently codified by the IEEE POSIX standard. The NFS protocol not only allows reading and writing of files, it also supports manipulation of directories.

Each NFS client system builds and maintains its own file system view. This view results from a hierarchical combination of its own file systems and the file systems of servers to which it wants access. At any given directory of this view, the client system may attach a new sub-tree of directories from an NFS server. This process of attaching a new sub-tree of directories is called mounting a remote file system. The directory to which the remote file system is mounted is called the mount point.

An important effect of the mount operation is that the files in the mount-point directory are no longer visible to the client. The newly mounted files in the remote file system are visible instead.

Another important principle is that NFS mounts that are made by a server, when it acts as a client to another system, are not visible to its clients. The clients see only the files that are physically located on the server.

For users of MVS and CMS, perhaps the most important aspect of the NFS design is its orientation toward being a network service instead of being the file system component of a distributed operating system. This orientation is critical in enabling the use of NFS on operating systems that are dissimilar to the UNIX environments in which NFS was originally implemented. The primary requirements for an operating system to participate in NFS are the ability to interpret a hierarchical file system structure and to share UNIX format user identification numbers. Other similarities to UNIX are not required. The SAS/C CSL NFS implementation is able to effect support for directories and UNIX user identification on MVS and CMS.

SAS/C NFS Client Overview

When working in a distributed environment without file sharing, the barrier between systems can become problematic. Files that are needed on one system often reside on another. The solution of transferring the entire file, using File Transfer Protocol (FTP) for example, is practical if the file is small and seldom changes, but becomes much more laborious when this is not the case.

Traditionally, programs running on MVS and CMS have had little or no access to files located on PCs, workstations, and other non-mainframe computers. The SAS/C CSL NFS client support changes this situation. For example, in the cross-development environment you can run the SAS/C Debugger on the mainframe while your source and debugger files reside on a workstation.

Accessing Files

The SAS/C CSL NFS client transient libraries enable a new filename style prefix, **path:**, in SAS/C filenames. In the same way that an MVS program can use the **dsn:** prefix to open a file by data set name, the program can now open an NFS file with the **path:** prefix. Thus, files that are accessed using NFS are placed in a separate name space from traditional MVS or CMS files. This separation is due to differences in file system organizations, such as directories versus partitioned data sets, rather than the fact that one group is local and the other is remote.

Mounting Directories

SAS/C CSL functions and configuration files are available to mount directories in the mainframe environment. As multiple mounts are established from one or more remote machines, the CSL NFS client library maintains a unified hierarchical view of the resultant directory structure. With the CSL NFS client, mounts are the responsibility of the individual user, not of a system administrator.

For example, a configuration file with the following line can be used if the user wants to access a UNIX root directory / on a machine named **acct.langdev.abc.com**.

```
acct.langdev.abc.com:/ / nfs
```

This indicates that the root of the **acct.langdev.abc.com** machine should be mounted as the root directory on the mainframe, thus enabling a debugger user to specify **set search** commands relative to the mount point. (See Chapter 9, "Cross-Debugging" for information about the SAS/C Debugger's **set search** command.)

To continue our example, suppose the user now invokes the debugger on the mainframe and enters the following **set search** command:

```
set search userinclude =
   "path:/usr/name/project/headers/%leafname"
```

The debugger will now look for user include files in the **/usr/name/project/headers** directory on the remote workstation named **acct.langdev.abc.com**.

In a more complicated setup, many different UNIX workstation file systems can be mounted together. The overall organization is the responsibility of the mainframe user, and the pathname for a particular file will often differ from what would be used on any of the systems individually.

File Security

The CSL NFS client enforces security controls that prevent unauthorized access to files on the server. Before the user can access an NFS file, the user identification must be authorized by the local RACF compatible security system, if one is available, and by a login server running on a UNIX system. If a local security system is available, this login process can be invoked automatically by the CSL library. If not, the user must supply a UNIX, or other NFS server operating system, username, and password.

In either case, the NFS client software maintains the standard UNIX, or POSIX, User Identification (UID) and Group Identification (GID) numbers for the duration of the user's session. The NFS client software controls access to remote files based on the user identification and the file's permissions.

Installation Considerations

The NFS client software depends on the SAS/C transient library, the SAS/C CSL transient library, and the TCP/IP software provided by your TCP/IP vendor. These must all be installed properly for the NFS client software to function correctly. Refer to SAS/C Library Reference, Third Edition, Volume 2 for additional information.

The NFS client commands must be accessible to users. On CMS, this involves accessing the disk. On MVS, the commands can be found if the commands are placed in linklist or LPALIB, or if they are in a data set allocated to the DDname CPLIB (provided that the optional SAS/C TSO command support is installed). Alternatively, MVS sites with REXX support can use REXX EXECs which invoke the commands. This avoids any need to install the SAS/C TSO command support.

In addition to mainframe installation considerations, you must coordinate NFS usage with the administrators of the NFS servers. They must grant the mainframe access in their configuration files. Additionally, they must install a login server for mainframe users to contact.

SAS/C CSL comes with distribution kits (in UNIX **`tar`** format) for two login servers. The first is the standard PCNFSD version 2 server from Sun Microsystems. The second is the CSL's **`sascuidd`** server, which is used for login without a password. If the NFS network is already running a PCNFSD version 1 server, it can be used instead of the PCNFSD version 2 server. The distribution kits come with "README" and "Makefile" files to guide the process of building the programs on your login server operating system.

PCNFSD may be hard to port to some systems, particularly systems that are not UNIX systems. There are a number of alternative approaches to solve this problem. If there is a secure UNIX system available in the network that is already running PCNFSD, then that system can be used. If no such system is available, sites with mainframe security systems can rely exclusively on **`sascuidd`** (which is much easier to port). **`sascuidd`** will run on any POSIX system that also supports RPC. It is also possible to use a stripped down version of PCNFSD. Only the authorization and null procedures are needed for CSL NFS. The others (mostly related to printing) are not needed.

Whatever server is installed and used, it must be up and running whenever mainframe users might need access to NFS files.

NFS Security Administration

The installation of NFS client software on any system should be a security concern for administrators of NFS servers. The availability of client software might enable file access by users to whom the access was not previously possible.

All security on NFS servers is via UNIX (or POSIX) UIDs and GIDs. The UID is a number that represents a user. The GID represents a group of users. The NFS design assures that all participating machines share the same UID and GID assignments. MVS users are identified by a security system such as RACF or ACF2. CMS users are identified by entries in a CP directory. UNIX UIDs and GIDs are not normally associated with mainframe users.

Administrators of NFS servers can usually control, on a file system basis, which client machines can access files via NFS. When security is a concern, the ability of the client machine to allow only authorized UID and GID associations is the most important factor. The CSL NFS client software derives its UID and GID associations from a combination of mainframe security system and UNIX servers. The exact source authorization depends on site configuration.

Because of differences between UNIX and mainframe operating systems, and because of a lack of reserved port controls in current mainframe TCP/IP implementations, the CSL NFS client software is generally less secure (in authorizing UID and GID associations) than most UNIX NFS client implementations. Methods of attack are briefly described in the SAS/C CSL installation instructions. Note that it is server file security that is of concern. An NFS client implementation can pose no additional security threat to files on the client (in this case the mainframe) unless it gives unauthorized access to files containing passwords. Note also that most UNIX NFS servers allow controls for which file systems can be accessed, thus limiting exposure to unauthorized UID associations.

Because it can use authorizations that are provided by a mainframe security system, CSL NFS client software is generally more secure than NFS client implementations on PC operating systems.

UID/GID Acquisition

The SAS/C CSL NFS client software will always retrieve the UNIX UID and GID information from a UNIX server. The retrieval of the UID information is based on a UNIX username. The association of UNIX username to UID/GID is always performed by a UNIX server.

One of two methods is used to associate a UNIX username with a mainframe user. If there is a (RACF compatible) security system installed, profiles can be established to authorize mainframe users to UNIX usernames. Users whose mainframe login ID is the same as their UNIX username are a special case that can be authorized using a single profile. There is considerable flexibility in this assignment. For example, the association between mainframe userids and UNIX usernames need not be one-to-one. When this method is used, the **`sascuidd`** login server is used to provide UID information for that username. No UNIX password is required.

A second method is for the mainframe user to supply the desired username and password to the UNIX login server PCNFSD (version 2 if available, otherwise version 1), which authorizes the username (based on the password) and supplies UID information in one step.

The first method will generally be preferred when available because it makes login easier and removes the requirement that UNIX passwords be present on the mainframe.

For TSO or CMS users, the UID information is stored in environment variables. The UID is stored in **`NFS_UID`**. The primary GID is stored in **`NFS_GID`**. The list of supplementary GIDs (**sascuidd** and PCNFSD V2 only) is stored in **NFS_GIDLIST**. **NFS_LOGINDATE** is set to the date of the login.

The environment variable NFS_LOGINKEY receives an encrypted value which is used by subsequent NFS calls to determine that these environment variables have not been tampered with.

NFS logins must be reissued each time the user logs in to TSO or CMS. Authorization is also lost after about 48 hours, even if the user does not log off. This prevents users from retaining their authorization indefinitely, even after they have had their UNIX authorizations removed.

At most 16 additional GIDs are allowed. This is the maximum supported by the PCNFSD protocol. A user who can login as UID 0 (root) will probably not be given full authority by the NFS server system. Most NFS servers remap UID 0 to a UID value of **(unsigned short) -2**.

SAS/C NFS client capabilities cannot be used until a successful login has occurred. Successive calls to NFSLOGIN can be made in order to access a different server or to use a different login ID. If a security system is present to allow login without a password, the actual login may be performed automatically when an NFS operation is requested. No corresponding logout is required.

When a mainframe security system is present, it can also control which login server a user is allowed to access. This prevents users from rerouting their login authorization request to a less trusted machine. It also reduces the risk of a user sending a UNIX password to a "Trojan horse" program that is running on an unauthorized system.

Use of a mainframe security system requires the definition of a generalized resource named LSNUID. The mainframe security administrator can then enter profiles that give mainframe users access to particular login servers and equate mainframe userids with UNIX usernames. The next section describes this in detail.

RACF Definitions for NFS Clients

The SAS/C CSL NFS Client mainframe security system interface is based on profiles that are defined for a generalized resource named LSNUID. Using this resource, you grant specific mainframe users access to specific UNIX userids and login servers in the same way that DATASET profiles allow you to grant users access to mainframe files.

Until this resource is defined and activated, the NFS client code behaves as it does when no security system is installed.

Here is a description of the macro parameters needed to define LSNUID (in a RACF environment).

```
ICHERDCE    CLASS=LSNUID
            ID=nn
            MAXLNTH=39
            FIRST=ANY
            OTHER=ANY
            POSIT= (prevented when RACF unavailable,
                    auditing if you want it,
                    statistics if you want them,
                    generic profile checking on,
                    generic command processing on
                    global access checking off)
ICHRFRTB    CLASS=LSNUID
            ACTION=RACF
```

In RACF, once this resource is defined, it must also be activated via the command:

```
SETROPTS CLASSACT(LSNUID)
```

The NFS client libraries make authorization inquiries about the following profile names (all requests are for "read" permission):

`LOCAL_USERID`
Users who are permitted to this profile are authorized to use their mainframe userid (lowercased) as the UNIX username without specifying a UNIX password.

`USER_name`
Users who are permitted to this profile are authorized to use the string name (lowercased) as their UNIX username without specifying a UNIX password. For example, if a mainframe user is permitted to the profile **`USER_BILL`**, then he is allowed to assume the UNIX username of **`bill`**.

`Pddd.ddd.ddd.ddd`
This specifies the network address (dotted decimal) of a PCNFSD server which the user can access to obtain a UID and GID. Permissions against servers prevent users from setting up unauthorized versions of PCNFSD on a less trusted machine and then directing their login queries to it. For example, if mainframe user BILL is permitted to P149.133.175.68, he can use the server at that IP address when logging in. Leading zeros are not allowed in these names. That is, the previous profile could not have been for P149.133.175.068.

`Sddd.ddd.ddd.ddd`
This is similar to the above, but permits access to a **`sascuidd`** server.

Configuring a Default Login Server

In most cases, it is better for users to reach a default login server. Having a correct default reduces user effort and confusion. But most importantly, the correct default must be set if the NFS client library is to perform logins automatically.

You can control the login server in three ways. One way is to set the NFSLOGIN_SERVER environment variable in the user's PROFILE EXEC or TSO startup CLIST. Another way is to apply the default login server configuration zap that is supplied in the installation instructions. The best method is to accept the default name nfsloginhost and to configure your nameserver or **`/etc/hosts`** format file accordingly.

Developing Standardized File-System Configurations

You may want to set up the file system configuration for users. If so, you can create a system-wide **`fstab`** file to perform their mounts. The search rules for the **`fstab`** file include a provision for a system-wide name. Users who do not set up **`fstab`** files of their own will get the system-wide file. If you want users to save file system context between programs, you can define the ETC_MNTTAB environment variable in the PROFILE EXEC or TSO startup CLIST.

Diagnosing Problems

The first step in identifying problems is to look carefully at the diagnostics produced by the debugger at the point where the failure occurred. Depending on whether the messages are generated by the debugger or by the library, the messages may be printed in the log window, or may be printed in line mode after erasing the debugger screen.

Many user problems are caused by incorrect installation of system software. These problems can often be diagnosed by understanding what is missing. Sometimes a configuration file is missing. Other times an environment variable definition is needed, or a REXX EXEC is not placed where it will be accessed.

In other cases, problems are caused by network and server failures. For server problems and failures on remote systems, the **RPCINFO** and **SHOWMNT** commands are useful. **SHOWMNT** is described in “SHOWMNT” on page 114, and **RPCINFO** is described in SAS Technical Report C-113, SAS/C Connectivity Support Library, Release 1.00. Both **SHOWMNT** and **RPCINFO** are compatible with the equivalent commands on UNIX.

For true network problems, SNMP or other network diagnostic facilities are most useful.

Recommended Reading

O’Reilly & Associates, Inc. publishes "Managing NFS and NIS," by Hal Stern. This book describes NFS administration in a UNIX environment. Many of the concepts and topics discussed may also help you administer mainframe NFS client software. If you cannot locate this book in your local bookstore, O’Reilly & Associates may be contacted at:

O’Reilly & Associates, Inc.

103 Morris Street, Suite A

Sebastopol, CA 95472

(800) 998-9938 US/Canada

707-829-0515 overseas/local

707-829-0104 Fax

NFS Administrator Commands

In addition to the commands described in Appendix 2, “Using the NFS Client,” on page 117, as an NFS administrator you should be familiar with the **SHOWMNT** command, which is described in “SHOWMNT” on page 114.

SHOWMNT

Queries an NFS server for file system information

SYNOPSIS

```
SHOWMNT [-e] [-d] [-a] [host]
```

DESCRIPTION

The **SHOWMNT** command queries an NFS server for information on file systems that may be mounted by NFS.

host is the hostname of the NFS server. If you omit this parameter, **SHOWMNT** returns information about the NFS server on the local machine (if one is installed).

SHOWMNT handles two basic types of lists. The first is an exports list. The exports list tells which file systems can be mounted. The second is a list describing which mounts have actually taken place. The form of the second list depends on the **-d** and **-a** flags.

The **-e** flag requests the exports list. This includes information about which hosts are authorized to mount the listed file systems. This information may either be everyone, or a list of group names that represent a set of hosts. If it is authorized, a host may mount any of the listed file systems.

You can use the following command when you are trying to determine the name of a file system to mount:

```
SHOWMNT -e
```

Note: You can often mount subdirectories of the listed file systems. Whether or not you can do this depends on whether the subdirectory is in the same physical file system on the server. Contact the server administrator or examine server configuration files to determine this. △

If **-e** is used in conjunction with other flags, this exports list will be printed first, followed by the list describing actual mounts.

If you don't specify any flags, **SHOWMNT** prints the list of actual mounts, showing only the names of the hosts that have a mount. The list is sorted by host name.

If you specify **-d**, **SHOWMNT** prints the list of actual mounts, showing only the names of directories that have been mounted. The list is sorted by directory name.

The **-a** flag gives the most verbose format for the list of actual mounts. It indicates that the list should be printed as "host:directory" pairs. If you do not use **-d**, **SHOWMNT** sorts the list by host. If you do use **-d**, **SHOWMNT** sorts the list by directory.

INVOCATION SYNTAX

The syntax is generally identical to that shown above. On MVS, system administration considerations may require use of the TSO CALL command or other techniques.

EXAMPLES

```
showmnt -e byrd.unx
```

Show mountable file systems on the byrd.unx NFS server.

```
showmnt byrd.unx
```

Show the list of other hosts that have mounted the NFS file system from byrd.unx.

APPENDIX 2
Using the NFS Client

Introduction

In a cross-development environment, the Network File System (NFS) client support provided by the SAS/C Connectivity Support Library enables the SAS/C Debugger to communicate with the host workstation. This appendix provides the basic information necessary to use this NFS support. Additional information is contained in Appendix 1, "Installing and Administering the NFS Client," on page 107 and SAS Technical Report C-113, SAS/C Connectivity Support Library, Release 1.00.

The NFS client feature provides flexibility in configuring NFS for each user. The degree of effort required to set up your configuration depends on the amount of support given by the system administration staff at your site.

For example, minimal user effort is required when the system administrators provide a centralized mount-configuration file and set up security-system definitions to allow automatic login. In this situation, users can begin specifying NFS filenames to application programs immediately. On the other hand, some sites may leave mounting files to the individual user. Lack of a RACF compatible security system might require that users issue an NFSLOGIN command at the beginning of each session. Even at sites where a centralized configuration has been set up, individual users with specialized access requirements may still develop their own configurations.

If your site management has already developed a configuration and makes it available to you automatically, perhaps with some instructions for using NFS at your site, you can bypass this section. Otherwise, you may need to use some of the commands and facilities described here.

Before using NFS to access remote files, you must understand two things:

- □ How to log on to an NFS login server.
- □ What remote files you want to access.

Logging on to the NFS Network

NFS servers use a UNIX, or POSIX, file-permission system. This system gives each user a UID, a GID, and possibly several additional supplementary GIDs. Each file is assigned ownership by user identification number (UID) and by group identification number (GID). Permissions for the file are set based on whether the user desiring access is the owner (has the same UID as the file), is in the file's group (has a GID that matches the GID of the file), or is some other user. For each of these three categories (owner, group, and other) read, write, and execute permissions can be assigned.

To access files using NFS, your session on MVS or CMS must acquire UID and GID numbers that correspond to some user on the NFS server network. You acquire these numbers by contacting a login server on the NFS network to ask permission to access files according to a username that is known to that server. In many cases, contact with the NFS login server can be automatic the first time you access an NFS file. In other cases, you must issue the NFSLOGIN command to effect the login.

The function of the login server is to check your identification and grant you access to the network. Once logged on, the login server functions as an NFS server and provides access to the files located on the machine on which it resides. At this point you may also use the network to access files controlled by other NFS servers on other machines.

If you have a RACF compatible security system running on your mainframe and your site administration has given you access to your NFS login server username, then the security system can vouch for you and no password is required. Note that the login server username is not necessarily the same as your MVS or CMS userid. If you do not have a security system, then you will need to enter your password during the login process.

In summary, the login process can involve three pieces of information:

- □ host name of the login server. (For example, the host name of a workstation running UNIX that acts as an NFS server.)
- □ login server username. (For example, your username on UNIX.)
- □ login server password for that username.

The requirement for a password depends on whether a mainframe security system can provide authentication for login server usernames. If the NFS client software can determine the other two pieces of information, either by default or by environment variables, then automatic login is possible. Otherwise, the NFSLOGIN command must be used.

For example, if your NFS network is composed of UNIX machines, your UNIX username is **`comkzz`**, and your login server is a UNIX machine called **`byrd.unx`**, then the CSL NFS client software must contact **`byrd.unx`** and provide **`comkzz`** as the user name. If your MVS username is also COMKZZ (the same except for upper case), the mainframe security administrator has authorized you to use the **`comkzz`** username for NFS, and **`byrd.unx`** has been configured as the default login server at your site, then the NFS client library will log you in automatically the first time you try to use NFS.

If, on the other hand, your site does not have RACF, a password is required. In this case, you need to issue the NFSLOGIN command to enter your password. See "NFSLOGIN" on page 124 for details.

After the login processing has succeeded, your session receives a UID and one or more GIDs. These control your subsequent accesses to NFS files.

Accessing Remote File Systems

Logging on establishes UID and GID information. The next step is to mount the remote file systems that you want to access.

Because the SAS/C CSL NFS client feature runs totally within your user address space under MVS, or on a virtual machine under CMS, you must mount remote file systems before accessing NFS files. A number of facilities are provided to make this process as transparent as possible. Mounts can occur in three ways:

- The configuration file, **`fstab`**, specifies a mount that occurs at session or program startup.
- You issue the MOUNT command.
- An application program performs a mount as part of its own processing logic.

At sites with standardized configurations, a series of mounts may be provided automatically. In this case, you do not need to do additional work unless you want a different configuration.

Saving File-System Context

Assuming that you are doing the configuration yourself, one of the first things to decide is the duration of your mounts. That is, do you want mounts and directory changes from one program to be preserved for the next program that is run? Mounts and directory changes form a file system context that may be restricted to the execution of a particular program or may be shared serially by programs under TSO or CMS.

Not sharing context can be easier. NFS mounts are very fast and involve minimal processing on the server. The serial sharing of file system context is accomplished using the **`mnttab`** file. When only a few file systems are mounted, reissuing the mounts in each program can be faster than reading and writing the **`mnttab`** file.

Unfortunately, processing the **`mnttab`** file at program startup and shutdown adds noticeable delays to otherwise fast commands and programs. The NFS sample programs **`cd`**, **`pwd`**, and **`ls`** illustrate this. Overall NFS performance is much better when a single program does many operations. Sharing is required, however, if working directory changes are to be preserved from one program to the next. You should always save the file-system context when working with the SAS/C Debugger in a cross-development environment.

You specify serial sharing of file system context by setting the ETC_MNTTAB environment variable to the name of a file to contain the context. For example, under TSO, you might use the value TSO:ETC.MNTTAB. This creates a file tsoprefix.ETC.MNTTAB. Under TSO you set the value using the PUTENV command. Allocating a DDname of ETCMNTTB has the same effect under MVS batch and may be more convenient. Under CMS, you can set the value using GLOBALV commands with the CENV group. Refer to SAS/C Compiler and Library User's Guide and SAS/C Library Reference, Volume 1, for more information about using environment variables with the SAS/C Compiler.

You do not need to create the **`mnttab`** file yourself. The NFS client library will create it automatically. It will also be deleted each time you log in to the NFS server. Note that, unlike the conceptually similar UNIX **/etc/mnttab** file, this file has a binary format. It also contains information, notably the current working directory, that is held by the kernel in UNIX.

Finally, be aware that the **mnttab** file cannot be shared simultaneously by many programs. If you are managing multiple programs that use NFS concurrently, either set up multiple **mnttab** files or set them up not to save context at all.

To avoid serial sharing, do not set the environment variable. Be aware that in this case, the MOUNT command and the sample cd command appear to have no effect, because the changes that they request are not saved when they end. When not sharing file system context, you will normally invoke all your mounts with the **fstab** configuration file.

Setting Up an fstab Configuration File

When NFS starts with no **mnttab** file available, either because there is no serial sharing of file system context, or NFS has not yet been used, the NFS client library searches for an **fstab** configuration file from which to perform initial mounts. The **fstab** file removes the need to issue mount commands manually each time NFS is used.

The **fstab** configuration file format is identical to that used on most UNIX systems. It should have a series of lines that specify mount points using the following format:

```
server : directory mount-point type options
```

Fields are separated by white space, and any fields that follow the options parameter are ignored. You can also include comments in the **fstab** configuration file. The pound (#) character at the beginning of a line or preceded by whitespace indicates that the rest of the line is a comment.

For NFS file systems, the device is specified as a server name followed by a colon (:), which is followed by the name of the directory to mount. This name must be a physical file on the server. It must not be a name that was created by NFS client features of the server. This is a common source of confusion. Users of the NFS server are often accustomed to specifying directory names that are not physical directories on their system. As discussed earlier, the design of NFS does not cause these names to be propagated automatically to NFS clients of that server.

The mount-point parameter must be a pathname in the directory hierarchy that is being created on the mainframe. In order for the first directory to be mounted, the mount point must be a slash (/), which indicates the root directory. Following NFS conventions, later mount points must be actual directories in a file system that have already been mounted. The directories that are being mounted then obscure the contents of the directory they are mounted on.

The type parameter must be **nfs**. As on UNIX, the table definition is generalized to accommodate multiple types of file systems; however, at present only NFS file systems are supported.

Mount options, which are described in “Mount Options” on page 121, generally are not needed.

Output A2.1 on page 121 shows a typical **fstab** configuration file:

Output A2.1 Example fstab configuration file

```
# My NFS setup
byrd.unx:/local/u/bill  /       nfs     #No mount options
server.unx:/tools       /tools  nfs ro  # Mount tools read-only
elgar.langdev:c:/       /lang   nfs     # Mount from OS/2
```

This example assumes that the **/local/u/bill** directory on **byrd.unx** contains subdirectories called **tools** and **lang**. Presumably these are empty directories that were set up to serve as mount points for the second and third mounts. If they are not empty, any contents that they have are obscured to the mainframe user by the second and third mounts. Instead of seeing the contents of the local directories, the corresponding directory trees from the **/tools** directory on server.unx and the **c:/** directory on **elgar.langdev** are seen by the mainframe user at those locations.

The **fstab** data set is located in the following manner:

1. If there is an environment variable named **ETC_FSTAB**, its value is used. Note that the default style is **ddn:**. Remember to include the style at the beginning of the name if you want a different one, such as in **tso:etc.fstab**.
2. On MVS, if there is a DDname of ETCFSTAB, it will be used.
3. The next data set in the sequence depends on the operating system you are working under.
   - Under TSO, *tsoprefix*.ETC.FSTAB is used.
   - Under MVS (other than TSO), if the userid can be determined, *userid*.ETC.FSTAB is used.
   - Under CMS, ETC FSTAB is used.
4. If on MVS, *zappedprefix*.ETC.FSTAB is used. The *zappedprefix* defaults to NFS if not zapped and can be overridden by the **NFS_PREFIX** environment variable.

The **fstab** data set cannot itself be accessed with the **path:** prefix. See "Accessing Files" on page 109 for information about the **path:** prefix.

Mount Options

Mount options control the operation of mounting the file system, as well as the file system's characteristics for subsequent use. They must be separated by commas, with no intervening spaces. They can be specified in either uppercase or lowercase. Mount options are not usually needed; the defaults are generally adequate.

Table A2.1 on page 121 contains the options you can specify.

Table A2.1 Mount Options

Option	Description
RW	Indicates that the file system is read/write. This is the default setting.
RO	Indicates that the file system is read-only.
DELTAMIN	Indicates the time adjustment in minutes to be applied to time stamps on the given file system. This can be useful when file systems are set to operate in different time zones. This value can be either positive or negative.

Option	Description
RETRY=*n*	Number of retries for mount failures. The default is 1. The parameter affects only mount attempts. It does not affect other operations such as read and write. (See RETRANS for other operations.)
RWSIZE=*nnK*	Read and write buffer size. The default is 4K. The maximum allowed is 1024K.
TIMEO=*n*	Controls the timeout interval in tenths of a second used between retransmission attempts. The actual timeout interval begins at n tenths of a second and is doubled for each retransmission. The default TIMEO value is 7. (See also RETRANS.)
RETRANS=n</ code>	Specifies the number of NFS retransmissions. The default is 4. The timeout is multiplied by 2 for each successive retransmission.
SOFT	Specifies that a transmission attempt should be abandoned after a complete set of retransmissions fails. This is the default.
HARD	Specifies that a transmission attempt should not be abandoned after a complete set of retransmissions fails. If HARD is specified, the retransmission process is started over again after each set of transmissions is completed.
TEXT	Perform ASCII or EBCDIC translation on all files. An ASCII-to-EBCDIC translation is performed when the file is read from the server, and an EBCDIC-to-ASCII translation is performed when the file is written to the server.
BINARY	Always leave data in untranslated, binary form.
XLATE	The name of a loadable translate table to be used for ASCII and EBCDIC translation in this file system. This translation affects data that are read and written. By default, NFS data are translated using the IBM code page 1047 standard. The table is built in much the same manner as SAS/C CSL RPC translate tables. (Refer to the description of the **xdr_string** function in SAS Technical Report C-113, SAS/C Connectivity Support Library, Release 1.00) The only difference is that you may choose any load module name and then specify it here. If you have created an L$NAEXDR table for RPC, you may specify it to get the same translations for NFS data as for RPC strings. The XLATE option does not affect pathnames, which are controlled by the RPC L$NAEXDR translate table if present. If the translate table is not present, use the code page 1047 standard.

The **TEXT** and **BINARY** mount options allow you to override the defaults, which are determined by the debugger when it accesses a file on the workstation. However, we recommend using them only in unusual situations. When using the SAS/C Debugger, the settings defined by the debugger are generally appropriate.

Mounting and Unmounting Manually

When you are saving your file system context between programs, you can manipulate your file system organization by using the **MOUNT** and **UMOUNT** commands. These commands are described later in this chapter.

Manipulating Files and Directories

Once you are logged in and have the remote file systems mounted into the directory structure that you want, you can begin to access files. In many cases you can do this through SAS/C programs that are not aware of NFS simply by specifying **path:** where you previously specified a local file name. This will work if the particular program that you are using allowed you to specify the style prefix. For example, CMS programs that let you access CMS Shared File System files using the **`sf:`** prefix will now allow you to access NFS file using the **`path:`** prefix. If the program uses the correct setting for text or binary processing when it opens files, text files will be translated from ASCII to EBCDIC automatically. If it does not, you can use the **`TEXT`** and **`BINARY`** mount options to override the program's decision.

Existing SAS/C programs can also remove, rename, and check accessibility of NFS files.

If you are not saving file system context, or if you are, but have not run a program to change the initial directory, you must use the full pathname (from the mainframe point of view) in order to access a file.

Programs that were developed using SAS/C CSL can access and manipulate the remote file systems more completely. They can create, delete, and list directories. They can work with hard and symbolic links. They can change or check the current working directory, and they can retrieve and change UNIX, or POSIX, file-status information.

The SAS/C CSL product contains many sample programs which can also be used as simple utilities. For example, there is a simple **`ls`** command that lists the files in a directory. There is an **`ncp`** command that can copy files between mainframe file systems and NFS file systems (and can be much quicker than getting into FTP). These are simple sample programs. They do not have the full features of their UNIX equivalents, but they are useful.

The following examples are distributed with the CSL run-time transients provided with the SAS/C Cross-Platform Compiler:

Table A2.2 Sample Programs

Example	**Description**
`cd`	Change the directory (requires an ETC_MNTTAB setting)
`ls`	List a directory (no wildcards)
`ncp`	Copy files between mainframe and NFS file systems
`pwd`	Print the working directory

NFS User Commands

The following commands are used primarily by users who are running NFS client applications:

Table A2.3 NFS User Commands

Command	**Description**
NFSLOGIN	Authorizes TSO or CMS users to access files via NFS.
MOUNT	Mounts remote NFS file systems into the NFS client file system structure.
UMOUNT	Removes a previously established mount.

The format used to invoke the **`NFSLOGIN`**, **`MOUNT`**, and **`UMOUNT`** commands is generally identical to that shown in the following reference information. On MVS, system administration considerations may require use of the TSO CALL command or other techniques. See your system administrator for details. See “MOUNT” on page 125, “NFSLOGIN” on page 124, and “UMOUNT” on page 126 for reference information.

NFSLOGIN

Authorizes TSO or CMS users to access files via NFS

SYNOPSIS

Format 1: **NFSLOGIN** [**-s** *server*] [**-u** *username*] [**-p** *password*] [**-n**]

Format 2: **NFSLOGIN -f**

DESCRIPTION

The **NFSLOGIN** command authorizes TSO or CMS users to access files via NFS. In some cases the NFS client software can determine the correct server and username without you specifying them. If a RACF compatible security system is installed, the site can define particular mainframe users as having access to specified UNIX userids without requiring a password. If no password is required, and if the other values are correct by default, you do not need to use this command. The login will occur automatically when you access the first NFS file or directory.

The **NFSLOGIN** command is provided for sites and situations where either a password is needed or the default server or username values must be overridden.

See “Logging on to the NFS Network” on page 118 for an introductory discussion of NFS login considerations. Also see“NFS Security Administration” on page 111 and the description of the **nfslogin** function in SAS Technical Report C-113, SAS/C Connectivity Support Library, Release 1.00 for more detailed information.

The **-f** option requests a full-screen display. This display has fields for entering the same information that can be specified on the command line. The full-screen option provides non-display password entry.

The server parameter is the host name of the login server that you want to contact. This may differ from the servers on which files are being accessed. The specified host must be running the appropriate login server software. See Appendix 1, “Installing and Administering the NFS Client,” on page 107 for details. You can usually omit this option because the site can set up a default at installation time. Note also that, when a security system is installed, the mainframe security administrator controls your access to login servers. Using an unauthorized server causes a RACF violation.

For username, enter your username on the NFS login server. This is often different from your MVS or CMS login ID. You do not need to specify a username if the USER environment variable is set to the desired name, or if your login server username is the same as your mainframe userid converted to lower case.

If you do not have a RACF compatible security system, or if you want to login as a username that is not associated with your RACF profile, use the **-p** option or the password field to specify your password on the login server. The mainframe security system (if present) can also control whether a password will or will not be allowed on your NFSLOGIN.

Note that the **-p** option requires a value. The **-n** option is required for the special case where the UNIX (or other login server operating system) system account has a null password. The **-p** and **-n** options are mutually exclusive. Not specifying either **-p** or **-n** indicates that the user expects the mainframe security system to authorize access to the login server username. The full-screen display also allows for the special case of a null password.

If the login attempt fails, **NFSLOGIN** prints a message describing the reason. Otherwise it prints a message indicating success. The login fails if the login server is not running on the NFS network.

Note that you don't need to log out from the login server; your UID and GID permissions will expire after you log off of TSO or CMS. If you want to access files under a different user name, you can issue **NFSLOGIN** again. A login will expire after two days. If you are connected to a session for several days, you will need to log in again.

EXAMPLES

```
nfslogin -f
```

Invoke the full-screen login panel.

```
nfslogin -u bbritten -p ocean
```

Log in to the default login server with username **bbritten** and password **ocean**.

MOUNT

Mounts remote NFS file systems into the NFS client file system structure.

SYNOPSIS

Format 1: **MOUNT** *server* :*directory mount-point* [*options*]

DESCRIPTION

The **MOUNT** command is one method of mounting remote NFS file systems into the NFS client file system structure on the mainframe. This command is useful only when you have configured your session to save file system context. Otherwise, the effect of the mount disappears when the **MOUNT** command completes.

The server parameter specifies the name of the NFS server on which the files are physically located. The directory is the name of the directory for the directory tree that you want to mount. It must be a physical filename on that server (it cannot be created by the server's NFS client software).

The mount-point parameter specifies the name of the mainframe NFS client directory on which the remote file system is to be mounted. For the first mount, this must be a slash (/). For subsequent mounts, it must be a valid pathname in the directory structure established by existing mounts.

The options string is not required. It specifies mount options for the file system. See "Mount Options" on page 121. The string of options must be separated by commas, with no intervening spaces.

You cannot mount on a directory that is already being used as a mount point. You must first unmount the existing file system with the **UMOUNT** command.

Be aware that mounts made by this command are preceded by mounts from any **fstab** file.

EXAMPLES

These examples assume that there is no **fstab** file and that file system context is being saved.

```
mount byrd.unx:/local/u/bill /
```

Mount **bill's** home directory on "**byrd.unx**" as the root directory on the mainframe.

```
mount server.unx:/tools /tools ro
```

Add the **/tools** directory from **server.unx** as a subdirectory and treat it as read-only.

UMOUNT

Removes a previously established mount

SYNOPSIS

Format 1: **UMOUNT** *mount-point*

DESCRIPTON

The **UMOUNT** command removes a previously established mount. This command is useful only when you have configured your session to save file system context. Otherwise, the effect of the unmount disappears when the **UMOUNT** command completes.

The mount-point parameter specifies a mainframe pathname to a directory from which a remote file system will be unmounted. The directory must have been used in a previous mount operation.

You cannot unmount the root directory. If you want to mount a totally different root directory, delete the **mnttab** file and then mount the new root directory. NFSLOGIN also deletes the **mnttab** file.

You cannot unmount a file system that has other directories mounted over it, or a file system containing your current directory. Attempting to do so results in the following message:

```
UMOUNT failed: file or record in use.
```

EXAMPLE

This example assumes that file system context is being saved.

```
umount /tools
```

Remove the file system that was previously mounted at **/tools**. If the file system mounted at / had any files in its **tools** subdirectory, these now become visible.

APPENDIX 3 ar2updte and updte2ar Utilities

Introduction

The utilities **`ar2updte`** and **`updte2ar`** transform an **`ar370`** archive into a file which 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 which are not acceptable as member names to IEBUPDTE. **`ar2updte`** offers a translation feature which 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.

CAUTION:

`ar370` archives are created and maintained only by **`ar370`** and **`updte2ar`**. The internal structures and the data these files contain are in EBCDIC format. **`ar370`** archives should never be modified or accessed in any way, other than through **`ar370`**. Similarly, IEBUPDTE input format data files are created only by IEBUPDTE and

ar2updte. The internal structures and the data these files contain are also in EBCDIC. △

ar2updte Syntax

The **ar2updte** utility is invoked with the following command:

```
ar2updte [options...] infile outfile
```

The options portion of the command line 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 can be separated from the option by white space, but need not be.

Note: The case of option characters is not significant, but case is significant for most option arguments. △

The following options are recognized by the **ar2updte** utility:

Table A3.1 ar2updte Options

Option	Description
-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.
-x	specifies that no character translations will be applied to the member names during the archive to IEBUPDTE conversion. 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. Unless **-x** is specified, default member translation rules are used. See the section "Default Member Translation Rules" on page 129 for details.

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.

Examples

The following examples show typical **ar2updte** command lines.

```
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 then convert all x's to the dollar sign.

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 the at symbol (@).
- □ 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 the pound sign (#), 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.

`001 Error opening input file, "[`*`filename`*`]".`
An attempt to open the file filename failed. Check all input files for validity and integrity.

`002 Error opening output file, "[`*`filename`*`]".`
An attempt to open the file filename failed. There may be a file system problem or failure.

`003 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.

`004 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.

`006 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.

`007 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.

`009 Option -"`***`option`*****`" needs to be followed by an argument.`**
`Correct usage: ar2updte [-x | -t` ***`c1:s1`*** **`[-t`** ***`c2:s2`*** **`...]]`** ***`filein fileout`***
The -"option" option must be followed by an argument.

`010 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.

`011 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.

`012 Unable to identify ar370 archive, "[`***`filename`*****`]".`**
An **`ar370`** archive can not be located from the filename specified in the command line. The input file in the command line must be a valid archive file.

`013 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.

`014 "[`***`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`**.

`015 File is not recognized as an archive.  Can not 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 it is an archive. If the archive has been modified by any utility other than **`ar370`** or **`updte2ar`** data could be lost or corrupted.

`016 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.

017 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.

018 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.

028 The number of aliases for the member "[*member name*]" exceeds 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.

029 Duplicate member name "[*member name*]" has been generated in 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.

030 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 MVS 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 MVS card-image partitioned data set using the MVS SAS System's PROC SOURCE. **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.

CAUTION:

ar370 archives are created and maintained only by **ar370** and **updte2ar**. The internal structures and the data these files contain are in EBCDIC format. **ar370** archives should never be modified or accessed in any way, other than through **ar370**. Similarly, IEBUPDTE input format data files are created only by IEBUPDTE and

ar2updte. The internal structures and the data these files contain are also in EBCDIC. △

updte2ar Syntax

The **updte2ar** utility is invoked with the following command:

```
updte2ar [options...] infile outfile
```

The options portion of the command line 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 can be separated from the option proper by white space, but need not be. Note that the case of option characters is not significant, but that case is significant for most option arguments.

The following options are recognized by the **updte2ar** utility:

Table A3.2 updte2ar Options

Option	Description
-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 </ para>	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** will have the same linking characteristics as an archive produced directly with **ar370**. If **-s** is omitted, then the archive symbol table will reference only the member names and aliases referenced by ./ control statements in the input file. An archive produced without **-s** will have 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.

Examples

The following examples show typical **updte2ar** command lines.

```
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 testlib.a
```

Create a new archive named **`testlib3.a`** using the IEBUPDTE input format file named **`test.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`**.

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.

`003 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 composed of 80-byte records.

`004 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.

`006 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.

`007 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 in the command line. Be sure the command line is formatted correctly.

`008 Argument following -a cannot be longer than 8 characters.`
`Correct usage: updte2ar [-l] [-s] [-a` ***`ending`*** **`] [-t`** ***`c1:s1`*** **`[-t`** ***`c2:s2...`*** **`]]`** ***`filein fileout`***
The **`-a`** option specified a suffix that was more than 8 characters.

`010 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, and **`-t`** c:s.

`011 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.

`019 Invalid name for symbol, "[`*`symbolname`*`]" specified in a SYMDEF control statement.`

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.

`020 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.

`021 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.

`022 Encountered EOF in continued SYMDEF card in file, "[`*`filename`*`]".`

An **`ar370`** SYM DEF 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.

`023 Unable to open IEBUPDTE file, "[`*`filename`*`]".`

An attempt to open the file filename failed. Check all input files for validity and integrity.

`024 Unable to open ar370 archive file, "`*`filename`*`".`

An attempt to open the file, filename, failed. There may be a file system problem or failure.

`025 Read of input file, "[`*`filename`*`]" failed.`

When attempting to read the input file, filename, **`updte2ar`** was unable to read 80 bytes. The IEBUPDTE utility requires the input file to be composed of 80-byte records. Check the input file for validity and integrity.

`026 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 be the cause. Make sure the space available for the output file is large enough to hold all the output.

`027 Error in seeking to offset in file, "[`*`filename`*`]".`

An error occurred when attempting to position to an offset in the file, filename.

APPENDIX 4 Redistributing SAS/C Product Files

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 (LDL) 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. The SAS/C Redistribution Package is available on all SAS/C supported platforms and may only be licensed by current SAS/C Compiler sites.

In the cross-development environment, the files that compose the SAS/C Limited Distribution Library are located on the host workstation or on the target mainframe, depending on how your site has licensed SAS/C software. The LDL files are located on the host workstation if your site has licensed the SAS/C Cross-Platform Compiler independently of any SAS/C mainframe software. The LDL files are located on the target mainframe if your site has licensed the SAS/C Cross-Platform Compiler and is also a licensed SAS/C mainframe customer.

The files that compose the SAS/C Redistribution Package are located on the host workstation that has a licensed copy of the SAS/C Cross-Platform Compiler installed.

Limited Distribution Library

LDL Files on the Host Workstation

If your site licensed the SAS/C Cross-Platform compiler independently of any SAS/C mainframe software, the LDL files are located on your host workstation. The LDL files are listed in **`./lib/mvs/sascindp/redist.txt`** if your mainframe target is MVS. For ESA mode CMS, the LDL files are listed in **`./lib/cms/sascindp/redist.txt`**. For 370 Mode CMS, the LDL files are listed in **`./lib/pcms/sascindp/redist.txt`**.

`redist.txt` is a complete list of all 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.

LDL Files on the Target Mainframe

If your site licensed the SAS/C Cross-Platform compiler and is also a licensed SAS/C mainframe customer, the LDL files are located on your target mainframe. The LDL files may be copied to tape by running one of the following jobs:

- Under MVS, run the JCL contained in **`sasc.cntl`** (DUMPRLDB).
- Under CMS, run the DUMPRLDB EXEC.

The files copied to tape by these 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 MVS, 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 Packages

This section lists the programs and libraries that comprise the SAS/C Redistribution Package for each SAS/C supported platform. This list is subject to change at any time.

In a cross-development environment, the SAS/C Redistribution Package is licensed on a cross-platform host basis. That is, the SAS/C Redistribution Package may only be licensed for the host workstation that has a licensed copy of the SAS/C Cross-Platform Compiler installed.

For more information about redistribution, have your SAS/C Support Consultant or Installation 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 licensing documents.

Note: All of the files specified in Table A4.1 on page 137 through Table A4.16 on page 142 are specified relative to the installation location for the SAS/C Cross-Platform Compiler. See your SAS/C Support Consultant or Installation Representative for the installation location used on your workstation. △

SAS/C Redistribution Package for AIX

Table A4.1 on page 137, Table A4.2 on page 137, Table A4.3 on page 138, and Table A4.4 on page 138 list the files that comprise the AIX (RS/6000) components of the SAS/C Redistribution Package:

Executables

Table A4.1 Redistributable AIX (RS/6000) Executables

File	Description
`host/r6x/bin/cool`	SAS/C **`cool`** pre-linker
`host/r6x/bin/clink`	SAS/C **`clink`** pre-linker
`host/r6x/bin/ar370`	SAS/C **`ar370`** archive utility
`host/r6x/bin/objdump`	SAS/C object file display tool
`host/r6x/bin/atoe`	SAS/C ASCII/EBCDIC translation tools. (The **`etoa`** program is a hard-link to **`atoe`**.)
`host/r6x/bin/sheller`	SAS/C C++ template utility
`host/r6x/bin/sascc370`	SAS/C **`cool`** front end

Man Pages

Table A4.2 Redistributable man Pages

File	Description
`man1/cool.1`	documents the SAS/C **`cool`** pre-linker
`man1/clink.1`	documents the SAS/C **`clink`** pre-linker
`man1/ar370.1`	documents the SAS/C **`ar370`** archive utility
`man1/objdump.1`	documents SAS/C object file display tool
`man1/atoe.1`	documents the SAS/C ASCII/EBCDIC translation tools. (The **`etoa`** program is a hard-link to **`atoe`**.)

ar370 Libraries for MVS

Table A4.3 Redistributable ar370 Libraries (MVS)

File	Description
`lib/mvs/libc.a`	Resident library
`lib/mvs/libspe.a`	MVS SPE library
`lib/libcxx.a`	C++ library
`lib/cics/libc.a`	CICS Resident library
`lib/cicsspe/libc.a`	CICS SPE Resident library

ar370 Libraries for CMS

Table A4.4 Redistributable ar370 Libraries (CMS)

File	Description
`lib/cms/libc.a`	Resident library
`lib/cms/libspe.a`	CMS SPE library
`lib/pcms/libc.a`	370–Mode Resident library
`lib/pcms/libspe.a`	370–Mode SPE library
`lib/libcxx.a`	C++ library
`lib/cics/libc.a`	CICS Resident library
`lib/cicsspe/libc.a`	CICS SPE Resident library

SAS/C Redistributable Package for SunOS

Table A4.5 on page 138, Table A4.6 on page 139, Table A4.7 on page 139, and Table A4.8 on page 139 list the files that comprise the SunOS (SPARC) components of the SAS/C Redistribution Package:

Executables

Table A4.5 Redistributable SunOS (SPARC) Executables

File	Description
`host/s4x/bin/cool`	SAS/C **`cool`** pre-linker
`host/s4x/bin/clink`	SAS/C **`clink`** pre-linker
`host/s4x/bin/ar370`	SAS/C **`ar370`** archive utility
`host/s4x/bin/objdump`	SAS/C object file display tool
`host/s4x/bin/atoe`	SAS/C ASCII/EBCDIC translation tools. (The **`etoa`** program is a hard-link to **`atoe`**.)

File	Description
`host/s4x/bin/sheller`	SAS/C C++ template utility
`host/s4x/bin/sascc370`	SAS/C **`cool`** front end

Man Pages

Table A4.6 Redistributable man Pages

File	Description
`man1/cool.1`	documents the SAS/C **`cool`** pre-linker
`man1/clink.1`	documents the SAS/C **`clink`** pre-linker
`man1/ar370.1`	documents the SAS/C **`ar370`** archive utility
`man1/objdump.1`	documents SAS/C object file display tool
`man1/atoe.1`	documents the SAS/C ASCII/EBCDIC translation tools. (The **`etoa`** program is a hard-link to **`atoe`**.)

ar370 Libraries for MVS

Table A4.7 Redistributable ar370 Libraries (MVS)

File	Description
`lib/mvs/libc.a`	Resident library
`lib/mvs/libspe.a`	MVS SPE library
`lib/libcxx.a`	C++ library
`lib/cics/libc.a`	CICS Resident library
`lib/cicsspe/libc.a`	CICS SPE Resident library

ar370 Libraries for CMS

Table A4.8 Redistributable ar370 Libraries (CMS)

File	Description
`lib/cms/libc.a`	Resident library
`lib/cms/libspe.a`	CMS SPE library
`lib/pcms/libc.a`	370-Mode Resident library
`lib/pcms/libspe.a`	370-Mode SPE library
`lib/libcxx.a`	C++ library

File	Description
lib/cics/libc.a	CICS Resident library
lib/cicsspe/libc.a	CICS SPE Resident library

SAS/C Redistributable Packages for HP-UX

Table A4.9 on page 140, Table A4.10 on page 140, Table A4.11 on page 141, and Table A4.12 on page 141 list the files that comprise the HP-UX components of the SAS/C Redistribution Package:

Executables

Table A4.9 Redistributable HP-UX Executables

File	Description
host/h8x/bin/cool	SAS/C **cool** pre-linker
host/h8x/bin/clink	SAS/C **clink** pre-linker
host/h8x/bin/ar370	SAS/C **ar370** archive utility
host/h8x/bin/objdump	SAS/C object file display tool
host/h8x/bin/atoe	SAS/C ASCII/EBCDIC translation tools. (The **etoa** program is a hard-link to **atoe**.)
host/h8x/bin/sheller	SAS/C C++ template utility
host/h8x/bin/sascc370	SAS/C **cool** front end

Man Pages

Table A4.10 Redistributable man Pages

File	Description
man1/cool.1	documents the SAS/C **cool** pre-linker
man1/clink.1	documents the SAS/C **clink** pre-linker
man1/ar370.1	documents the SAS/C **ar370** archive utility
man1/objdump.1	documents SAS/C object file display tool
man1/atoe.1	documents the SAS/C ASCII/EBCDIC translation tools. (The **etoa** program is a hard-link to **atoe**.)

ar370 Libraries for MVS

Table A4.11 Redistributable ar370 Libraries (MVS)

File	Description
`lib/mvs/libc.a`	Resident library
`lib/mvs/libspe.a`	MVS SPE library
`lib/libcxx.a`	C++ library
`lib/cics/libc.a`	CICS Resident library
`lib/cicsspe/libc.a`	CICS SPE Resident library

ar370 Libraries for CMS

Table A4.12 Redistributable ar370 Libraries (CMS)

File	Description
`lib/cms/libc.a`	Resident library
`lib/cms/libspe.a`	CMS SPE library
`lib/pcms/libc.a`	370-Mode Resident library
`lib/pcms/libspe.a`	370-Mode SPE library
`lib/libcxx.a`	C++ library
`lib/cics/libc.a`	CICS Resident library
`lib/cicsspe/libc.a`	CICS SPE Resident library

SAS/C Redistribution Packages for Windows 95 and Windows NT

Table A4.13 on page 141, Table A4.14 on page 142, Table A4.15 on page 142, and Table A4.16 on page 142 list the files that comprise the Windows 95 and Windows NT components of the SAS/C Redistribution Package:

Executables

Table A4.13 Redistributable Windows 95 and Windows NT Executables

File	Description
`host\wnt\bin\cool.exe`	SAS/C **`cool`** pre-linker
`host\wnt\bin\clink.exe`	SAS/C **`clink`** pre-linker
`host\wnt\bin\ar370.exe`	SAS/C **`ar370`** archive utility
`host\wnt\bin\objdump.exe`	SAS/C object file display tool
`host\wnt\bin\atoe.exe`	SAS/C ASCII/EBCDIC translation tools.

File	Description
`host\wnt\bin\etoa.exe`	SAS/C EBCDIC/ASCII translation tools.
`host\wnt\bin\sheller.exe`	SAS/C C++ template utility
`host\wnt\bin\sascc370.exe`	SAS/C **`cool`** front end

Man Pages

Table A4.14 Redistributable man Pages

File	Description
`man1\cool.1`	documents the SAS/C **`cool`** pre-linker
`man1\clink.1`	documents the SAS/C **`clink`** pre-linker
`man1\ar370.1`	documents the SAS/C **`ar370`** archive utility
`man1\objdump.1`	documents SAS/C object file display tool
`man1\atoe.1`	documents the SAS/C ASCII/EBCDIC translation tools.
`man1\etoa.1`	documents the SAS/C EBCDIC/ASCII translation tools.

ar370 Libraries for MVS

Table A4.15 Redistributable ar370 Libraries (MVS)

File	Description
`lib\mvs\libc.a`	Resident library
`lib\mvs\libspe.a`	MVS SPE library
`lib\libcxx.a`	C++ library
`lib\cics\libc.a`	CICS Resident library
`lib\cicsspe\libc.a`	CICS SPE Resident library

ar370 Libraries for CMS

Table A4.16 Redistributable ar370 Libraries (CMS)

File	Description
`lib\cms\libc.a`	Resident library
`lib\cms\libspe.a`	CMS SPE library
`lib\pcms\libc.a`	370-Mode Resident library
`lib\pcms\libspe.a`	370-Mode SPE library

File	Description
`lib\libcxx.a`	C++ library
`lib\cics\libc.a`	CICS Resident library
`lib\cicsspe\libc.a`	CICS SPE Resident library

APPENDIX

5

Compatibility Notes

Changes for Release 6.50

The following section describes changes for Release 6.50 of the SAS/C Cross-Platform Compiler.

Marking and Detecting Previously Processed cool Objects

In Release 6.50, 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`**

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 you want this type of behavior from **`cool`**, the options can be specified such that the output object's decks are not marked and/or that such marking is ignored.

Changes for Release 6.00

Release 6.00 of the SAS/C Cross-Platform Compiler has new compiler option names and a different default pre-linker program. The following sections describe these changes and discusse their compatibility with previous releases.

Compiler Options

Prior to Release 6.00, most C parsing phase (phase `1`) compiler options used the syntax:

```
-W1,-option_name
```

where option_name was a mnemonic for the corresponding mainframe compiler option. In this release, these options have been replaced with options of the form:

```
-Koption_name
```

where option_name more closely resembles the corresponding mainframe compiler option.

Options for several other compilation phases have also been replaced with **-K**option_name forms, including:

- □ code generation phase (phase **2**)
- □ global optimizer phase (phase **g**)
- □ C++ translation phase (phase **C**)

Several non-phase-related options have also been renamed to indicate more accurately what the options are for.

Table A5.1 on page 146 shows the correspondence between the old option names and the new option names. To maintain compatibility with existing build procedures, the cross-platform compiler accepts the old names. However, we recommend that you migrate to the new names. For example, you can use either of the following commands and achieve the same results:

sascc370 -Kredef -Kcomnest alpha.c *(new syntax)*

sascc370 -W1,-cr -W1,-cc alpha.c *(old syntax)*

These commands compile **alpha.c** and allow redefinition and stacking of **#define** names, and nested comments. For a complete description of these compiler options, see "Option Descriptions" on page 47.

Table A5.1 Compiler Option Changes in Release 6.00

Old Option	New Option	Description
-W1,-ao	**-Kasciiout**	Character string constants are output as ASCII values.
-W1,-cc	**-Kcomnest**	Allow nested comments.
-W1,-cg	**-Ktrigraphs**	Enable translation of ANSI standard trigraphs.
-W1,-co	**-Kppix**	Allow nonstandard token-pasting.
-W1,-cr	**-Kredef**	Allow redefinition and stacking of **#define** names.
-W1 -cs	**-Kstringdup**	Create a single copy of identical string constants.
-W1,-hs	**-Knohmulti**	Specifies that system include files will only be included once.
-W1,-hl	**-Knoimulti**	Specifies that local include files will only be included once.
-W1,-i	**-Kindep**	Generate code that can be called before the run-time library framework is initialized or code that can be used for interlanguage communication.
-W1,-k	**-Ksmpxivec**	Generate a CSECT with a unique name of the form sname@. in place of @EXTVEC# (for SMP support).
-W1,!l	**-Knolineno**	Disable identification of source lines in run-time messages produced by the SAS/C Library.
-W1,-ll	**-Kstrict**	Enable an extra set of warning messages for questionable or nonportable code.
-W1,-q002=*filename*	**-Klisting=*filename***	Specify the name of the listing file.

Old Option	New Option	Description
`-W1,-v`	**`-Kvstring`**	Generate character string literals with a 2-byte length prefix.
`-W2,-q001=`*`filename`*	**`-Ksrcis=`*`filename`***	Override the name of the source file in the debugging file.
`-W2,-q003=`*`filename`*	**`-Kdebug=`*`filename`***	Generate a **`.dbg370`** debugging information file and, optionally, specify the full name of the file.
`-W2,-q004`	**`-Ksingleret`**	Forces the code generator to generate a single return sequence at the end of each function.
`-W2,-q006`	**`-Knodbgcmprs`**	Do not compress debugging information.
`-Wg,-!inline`	**`-Knoinline`**	Disable all inlining during the optimization phase.
`-Wg,-!inlocal`	**`-Knoinlocal`**	Disable inlining of single-call, static functions during the optimization phase.
`-WC,-wE`*`n`*	**`-w~`*`n`***	Cause warning message n to be treated as an error condition.
`-WC,-wM`*`n`*	**`-w+n`**	Specify that warning number n should not be suppressed.
`-WC,-wS`*`n`*	**`-w`*`n`***	Suppress warning message number n.
`-Knonuinc`	**`-Knousearch`**	Specify **`#include`** file search rules that are not typical of UNIX.
`-se`	**`-Kexclude`**	Omit listing lines that are excluded by preprocessor statements from the formatted source listing.
`-sh`	**`-Khlist`**	Print standard header files in the formatted source listing.
`-si`	**`-Kilist`**	Print the source referenced by the **`#include`** statement in the formatted source listing.
`-sm`	**`-Kmaclist`**	Print macro expansions in the formatted source listing.
`-ss`	**`-Ksource`**	Output a formatted source listing of the program to the listing file.
`-sx`	**`-Kxref`**	Produce a cross-reference listing.

Pre-Linker

In this release, the program **`cool`** replaces **`clink`** as the default object code preprocessor. If you do not suppress pre-linking with the **`-c`** compiler option, **`sascc370`** and **`sasCC370`** pre-link the object file with **`cool`**.

The **`cool`** program is designed to be backwards compatible with source code that was developed prior to Release 6.00. In addition to accepting all of the driver options supported by the old **`clink`** program, **`cool`** accepts the following options, which are new for this release:

- The **`-r`** option suppresses copying the run-time constants CSECTs to the output object file.
- The **`-s`*`nn`*** option defines the number of lines per page in the listing file.
- The **`-vo`** option creates only the EXTVEC# CSECT.
- The **`-xt`** option invokes a user exit program with optional data.
- The **`-yl`** option causes input control statements to be echoed to the listing.
- The **`-yg`** option includes "gathered" symbols in the listing.

- The **-yp** option includes a pseudoregister map in the listing.
- The **-zc** option allows processing to continue even if a corrupted **ar370** archive is detected.
- The **-zd** option allows multiple input files to define the same SNAME.
- The **-zi** option processes data after an INCLUDE statement in an input file.
- The **-zv** option prints additional informational messages.

Since **clink** is still distributed with the SAS/C Cross-Platform Compiler and C++ Development System, you can pre-link your program with **clink**, instead of **cool**, if desired. You can use either of the following methods:

1 Use the **-Kuse_clink** compiler option to invoke **clink** automatically when you run **sascc370** or **sasCC370**. For example, the following commands compile **alpha.c** and pre-link the output file with **clink**:

sascc370 -Kuse_clink -Anolineno alpha.c *(new syntax)*

sascc370 -Kuse_clink -Wl,-d alpha.c *(old syntax)*

You can pass any of the pre-linker options described in Table 6.1 on page 76 to **clink**, except the ones listed above, which are supported only by **cool**.

In this release, the recommended way to pass an option to the pre-linker during compilation is with the **-A***option_name* compiler driver form. In the first example above, the **-Anolineno** option is passed to the pre-linker to delete the line number and offset table CSECTs. However, for compatibility with existing build procedures, you can also specify the compilation phase with the **-W***l* prefix. In the second example above, the **-Wl**,**-d** option is passed to the pre-linker, which has the same effect as specifying **-Anolineno**. Notice that you must use the actual pre-linker option (in this case **-d**) when specifying the compilation phase with **-W***l*. For more information about specifying the compilation phase, refer to Chapter 3, “Compiling C and C++ Programs,” on page 39.

2 You can also suppress pre-linking when you compile your program and then call **clink** directly. For example, these commands compile **alpha.c** and pre-link the object file with **clink** in a separate step:

```
sascc370 -c alpha.c

clink -d alpha.o /libdir/libc.a
```

You can specify any of the pre-linker options described in Table 6.1 on page 76, except the ones listed above, which are supported only by **cool**. If you specify any **cool**-only options, they are ignored.

Index

Your Turn

If you have comments or suggestions about SAS/C Cross-Platform Compiler and C++ Development System 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

Your Turn

If you have comments or suggestions about SAS/C Cross-Platform Compiler and C++ Development System 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