Quick Start to C Using the SAS/C Mainframe Compiler
Dee Stribling, SAS Institute Inc., Cary, NC

ABSTRACT
This paper suggests ways you can use SAS/C documentation to enhance SAS/C program development. The paper begins with an overview and orientation to the documentation. After you're familiar with what's available, you will follow the development of a C program in terms of that documentation. The example program provides the basis for discussing how and where many SAS/C features, libraries, options, and so forth used in the program's development are documented. Knowing your way around the documentation can help you take advantage of features such as the optimizer and debugger, resulting in more efficient SAS/C applications.

This paper is directed towards new users of the SAS/C Compiler under either MVS or CMS. It is assumed that you have a basic understanding of the C language at the application programming level. The focus is on the documentation accompanying the SAS/C Compiler and the compiler rather than the C language itself.

INTRODUCTION
The SAS/C Compiler is a portable implementation of the high-level C programming language. The primary elements of the SAS/C Compiler are the compiler and the run-time library (a collection of routines that carry out operations not performed by the compiled code, such as mathematical functions and input/output). However, the compiler product also includes a number of utility programs, as well as several configurations of the run-time library for specialized environments.

The compiler also provides a complete library of C functions, including functions compatible with the ANSI Standard, functions related to the Standard, and functions that support non-ANSI features such as interuser communication.


This paper focuses on the first four books, as these form the basic reference set for both new and experienced users. The primary purpose of this paper is to explain how you can use this basic set to begin running programs using the compiler.

Note, however, that the compiler includes many features that are applicable to advanced C programming applications, including systems programming. These features, such as communication with assembler programs and SPE (Systems Programming Environment), are also extensively covered in the documentation set. Suggestions for how you can reference this material and the material in the last five publications are noted in the final section of this paper. However, since the intent of this paper is to provide a basic "view" of the documentation for someone new to the compiler with a basic knowledge of the C language, these topics are not covered here.

Information is presented in the following sections:
- Understanding the Documentation
- Using the Documentation in Program Development
- Getting Started with the Compiler
- Pointers to Efficiency Tips

UNDERSTANDING THE DOCUMENTATION
You have a C application and access to the SAS/C mainframe compiler. What is the first thing you do? Review the documentation that accompanies the compiler. At a minimum, you should read the introduction (Chapter 1) to each book in the basic set. The following sections provide an introduction to these four books. This overview should help you become acquainted with where different types of information important to using the compiler are located.

SAS/C Compiler and Library User's Guide
The SAS/C Compiler and Library User's Guide provides a functional description of the SAS/C Compiler. This book is a user-oriented reference to the SAS/C language and compiler implementation as well as a reference for compiling, linking, and executing C programs under TSO, CMS, and MVS-batch. The book includes usage, tutorial, and reference information, as well as a function and header file index and a glossary. There are also extensive examples, tables, and figures to help explain and summarize information. Important chapters for the new user at a basic applications level are:
- Chapter 1 - "Introduction"
- Chapter 3 - "Language Definition"
- Chapter 4 - "Compiler Processing and Code Generation Conventions"
- Chapter 5/6/7 - One of these, depending on mode (TSO, CMS, MVS-batch)
- Chapter 8 - "Compiler Options"
- Chapter 9 - "Run-Time Argument Processing"
- Appendix 1 - "Compiler Messages"
- Appendix 2 - "CLINK Messages"
- Glossary

Descriptions of each chapter in this basic set follow.
Chapter 1
Chapter 1, "Introduction," is important as it provides an introduction to the compiler, explains the organization and purpose of the book, provides additional references, specifies typographical conventions, and notes procedures for technical support.

Chapter 3
Chapter 3, "Language Definition," covers the C language as accepted by the SAS/C Compiler, including SAS/C extensions to the C language. The chapter also includes a section, following the format of the ANSI Standard, specifying how the Institute has chosen to implement some aspects of the language. This information provides important details you need to be aware of in programming, such as the number of significant initial characters in an identifier.

In addition to discussions of language elements (how pointer conversion is handled, for example) and language extensions (such as using the call-by-reference operator), several tables are provided to summarize information for quick reference. Included are tables covering the following:

- default representation for special characters
- integral type sizes
- escape sequence values.

Chapter 4
Chapter 4, "Compiler Processing and Code Generation Conventions," covers how the compiler implements certain aspects of processing and code generation. The compiler is implemented on machines with IBM 370 architecture running MVS, MVS/XA, MVS/ESA, CMS, or Bimodal CMS operating systems. This chapter focuses on hardware and operating-system-dependent features of the compiler associated with this architecture and operating system set.

An example of operating-system-dependent information is the syntax for specifying include files under MVS or CMS. An example of hardware-dependent information is how the compiler implements arithmetic data types. For example, doubles and long doubles are both 8 bytes in length.

Additional information on multibyte character processing, special pointer types, external names and features such as optimization and the __inline keyword is also included in this chapter. A special section is devoted to efficient programming using the SAS/C Compiler, noting several ways that you can improve your program's performance. (The last part of this paper discusses some of these sections.)

Important tables provided for the new user include the following:

- data type characteristics
- compiler return codes.

Chapters 5, 6, and 7
These chapters explain how to compile, link, and run your SAS/C program under TSO, CMS, and MVS-batch. All three chapters include examples of invoking the compiler and the syntax for specifying information such as library options and program parameters. Pertinent information from these chapters is covered in detail later in this paper under "Getting Started with the Compiler.

Chapter 8
Chapter 8, "Compiler Options," explains how you can modify the way your program is compiled. Chapter 8 provides information on the numerous compiler options provided by the SAS/C Compiler.

These options enable you to alter the way code is generated, change the appearance of listing files, and modify other aspects of compilation. This chapter explains what options are available and how to specify them under each environment (TSO, CMS, or MVS-batch). For example, you can choose to generate a listing of options specified for a compilation, a formatted source listing, cross-reference listing, and so forth. Each option is discussed in detail, and a summary table is provided for quick reference.

The section "Getting Started with the Compiler" later in this paper shows how using some of these options can affect your program compilation.

Chapter 9
Chapter 9, "Run-Time Argument Processing," explains how program arguments can be transferred from the external environment to the program. The SAS/C Compiler allows you to do this in two ways: by using the argv or argc parameters to main or by environment variables. In addition, the chapter also covers how to select certain run-time library parameters including function linkage, debugger invocation, and message generation. Redirection of stdin and stdout is also explained.

Details and examples are provided for specifying run-time arguments as well as a quick reference table summarizing available run-time options. (There are four types of run-time arguments. When run-time options (library options controlling program execution) are specified on the command line, they become one type of run-time argument.)

In summary, key areas from the User's Guide for the new user are

- Chapters 1, 3, 4 - provide an overview of the SAS/C Compiler and the compiler implementation of the C language
- Chapters 5, 6, 7 - document how to run the compiler depending on your method of operation (TSO, CMS, MVS-batch)
- Chapters 8 and 9 - discuss compiler options and run-time arguments
- Appendix 1 and 2 - discuss compiler and CLINK messages
- Glossary - provides explanation of key terms and concepts.

SAS/C Library Reference, Volumes 1 and 2
The two-volume reference set for the SAS/C Library focuses on describing the functions provided with the SAS/C Compiler, as well as additional background and tutorial material.

The SAS/C Library conforms to the ANSI Standard. A program that uses ANSI library functions correctly can be easily ported to other ANSI-conforming implementations. To help you tailor your program to your particular needs (portability may not be a concern, for example) the Reference volumes are organized as follows.

Volume 1 describes features that are either ANSI Standard, or related to Standard features, such as character- and string-handling functions. (SAS/C extensions to Standard features are not portable and are noted as such in the text.) In the first volume, the focus is on general C function groups: mathematical functions, memory allocation functions, input/output functions and so forth.

Volume 2 describes features that are not related to the Standard and are not portable. Volume 2 covers items such as the REXX SAS/C interface and low-level input/output functions. You should browse Volume 2 to see what is covered, but for the most part, the functions used in basic C applications are found in Volume 1.
Chapters 1 and 2 follow the same format in both volumes. Chapter 1 covers the volume's purpose and typographical conventions and contains a brief description of the contents of each chapter.

Chapter 2 provides important background information on using the library. The first section covers changes and enhancements since the last release, information on header files needed for the functions, and standard and non-standard type and macro definitions. Final sections in Chapter 2 provide summaries of function implementation, rules for using different releases of the compiler and library, and ABEND codes. Also included in Chapter 2 are several tables that provide quick summary and reference information.

Format for Function Descriptions
Both volumes also use the same format for function descriptions. This ensures consistency and provides a good anchor for quick reference via the function index. The format provides the following information about each function:

- Name
- Purpose
- Synopsis
- Description
- Return Value
- Caution
- Portability
- Example
- Additional references.

Like the User's Guide, both volumes contain valuable information for users of all levels of experience and interest. However, in addition to Chapters 1 and 2, the following material in Volume 1 provides a good starting point for new users:

- Chapter 3 - "Character Type Macros and Functions"
- Chapter 4 - "String Utility Functions"
- Chapter 9 - "Memory Allocation Functions"
- Chapter 12 - "I/O Functions"
- Appendix 1 - "Function Header Files and Compatibility"
- Appendix 4 - "Function Prototype Summary"
- Function Index.

SAS/C Source Level Debugger User's Guide
The SAS/C Source Level Debugger is a tool for checking and testing programs compiled with the SAS/C Compiler. The SAS/C Debugger (supplied with the compiler) runs interactively under CMS and TSO, and in batch mode under MVS. By using the debugger commands and command arguments, you can obtain information about what is happening at various points in an executing program. This document describes the features available with the debugger and also provides a tutorial to help you start using the debugger.

The information about the SAS/C Debugger is organized into five parts:

1. Introduction
2. Debugger Commands and Their Uses
3. Running the Debugger
4. Debugger Command Reference
5. Appendices

If you are just getting started with SAS/C programming, you should read the Introduction, and then focus on Part 2, the tutorial portion. Once you are familiar with how to use the debugger, you can continue to Part 3, covering compiler options related to running the debugger and instructions as to running the debugger in your particular environment.

Now that you're more familiar with what's available in the basic documentation set, the next two sections address the questions

- How can the documentation help me write a SAS/C program?
- How can the documentation help me run a SAS/C program?

Suggestions for answers to the first question are provided by building an example program in the next section, "Using the Documentation in Program Development." The second question is addressed in the last two sections, "Getting Started with the Compiler" and "Pointers to Efficiency Tips."

USING THE DOCUMENTATION IN PROGRAM DEVELOPMENT
In this section you'll see how you can refer to the documentation for assistance in writing SAS/C programs. Topics include an overview of the application used for the example, using reference aids in the documentation, and writing the final program. Unless otherwise specified, all references to the User's Guide in this section refer to the SAS/C Compiler and Library User's Guide.

Example Overview
Suppose you wanted to write a program that makes secret quotes, that is, more or less famous sayings in an encrypted alphabet. For good measure, each quote is encrypted using a different scheme. To keep things simple, input is a file of quotes, one per line. Output consists of a file of encrypted quotes, one per line. The terminal is stdin and stdout. Completion codes will be either 0 (success) or 8 (I/O read/write error). For good measure, you could name the program SQUOTE.

Processing consists of getting a quote, encrypting the quote, and displaying the encrypted version. To do this, the following tasks are needed:

- control the processing
- read the quote
- encrypt the quote
- write the quote.

The pseudocode for this can be sketched as

```c
main() /* Control the process */
{
    eof = GetQuote(quote); /* Read the first quote */
    while not eof /* If no more, stop */
    {
        Encrypt(quote) /* Call the encryption function */
        PutQuote(quote) /* Write to the output file */
        eof = GetQuote(quote); /* Read another quote */
    }
}
```
With a good idea of what the main() function does, you can begin writing the other functions, filling in declarations and definitions needed.

Regardless of where you start with the program, in the documentation, you should start with the index. (If you weren’t already familiar with which book to choose for certain kinds of information, you’d start with the table of contents or master index.) Starting at the function level, you have two reference choices in SAS/C Library Reference: the regular index (by name or task) or the function index. With this in mind, you’re ready to code the functions.

The Encryption Function (Encrypt)
The encryption needed for this function involves translating a string of characters from one form into another. String handling functions are common in the C language, so you know to look in Volume 1 of the Reference books. Looking in the index under “strings,” there’s an entry for “translation to another character set.”

Following this reference leads to the strxlt function description. This function translates one null-terminated character string to another. By looking at the SYNOPSIS and EXAMPLE sections and reading the function description, you can write the code necessary for the call. From the synopsis you can also make note of header files that are needed to support the function as well as add declarations for relevant variables and constants. The synopsis is

SYNOPSIS

#include <cstring.h>

char *strxlt(char *src, const char *table);

Within the function description for strxlt, a reference is made to the xlttable function. This gives you a lead to the function you can use to build the character translation table needed by strxlt. Throughout the reference volumes, related functions are always noted in both the description and the section titled SEE ALSO.

The code necessary for the call can be sketched in as follows:

static void Encrypt(char *quote) {
    char alphabet[ ] = ALPHABET; /* Uncrypted alphabet */
    char table[256]; /* Translation table */
    /* Encrypt routine goes here */
    strxlt(quote, table);
}

The encryption routine centers around a random number generator. A function for this, rand, can be quickly found in the function index (alphabetical list by function name) of Volume 1. Adding the code for this routine, the completed function becomes

static void Encrypt(char *quote) {
    int i, j;
    char alphabet[ ] = ALPHABET; /* Uncrypted alphabet */
    char table[256]; /* Translation table */
    /* Make a random encryption table. If the read function */
    /* returns an odd number, swap two letters in the alphabet */
    for (i = 0; i < 52; i++)
        for (j = 1; j < 52; j++)
            if (rand() % 2)
                alphabet[j] = alphabet[i];
                alphabet[i] = alphabet[j];
    xlttable(table, ALPHABET, alphabet);
    strxlt(quote, table);
    /* Encrypt the quote */
}

The next section explains how to read the quote to be encrypted and display the results.

GetQuote and PutQuote
These functions involve calls to SAS/C input and output functions. Whether you are new to the C language or just a new user of the SAS/C Compiler, you should read Chapter 12, "I/O Functions," in SAS/C Library Reference, Volume 1. Like other chapters introducing a related group of functions, this chapter provides a great deal of background and introductory material in addition to instructions on how to implement the function calls. If you are not familiar with a term used in the explanation, you can check the glossary (in the User’s Guide) for a definition.

Reviewing this chapter is important if you are developing a new program or porting a program to the SAS/C environment. The SAS/C Library provides several different I/O techniques to meet the needs of different applications. To achieve the best results, you need to be able to make an informed choice about which technique to use. To make these choices, you need to understand general C I/O concepts as well as I/O types and structures supported by MVS and CMS. To help you choose your program’s I/O functions, the chapter contains information in three sections:

- Technical Background (discussions of different I/O models)
- Technical Summaries (detailed discussion of C I/O components, such as opening files, file positioning, and standard files)
- Function Descriptions (detailed descriptions as noted earlier)

Once you’re familiar with what I/O functions are available in the SAS/C Library, you can check the function descriptions for the basic functions fgets, fputs, and ferror and code the two routines.

Completed Example and a Documentation Note
The previous descriptions illustrated two basic views of the documentation useful to program development. In the encryption routine, the information needed was located directly via the main index and function index. With the I/O functions, the documentation served as a source of background and conceptual information as well as a quick reference. Regardless of the view, reference or conceptual, the documentation is designed to provide you with the opportunity for both.

The completed program appears as follows:

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

#define MAX_QUOTE_LENGTH 100 /* Maximum length of a quote */
#define ALPHABET = AaBbCcDdEeFfGgHhIiJjKkLmMnNoOoPpQqRrSsTtUuVvWwXxYyZz

void main() {
    int eof;
    char quote [MAX_QUOTE_LENGTH+2]; /* Quote buffer */

    eof = GetQuote(quote); /* Read the first quote */

    while (true) {
        Encrypt(quote); /* Call the encryption function */
        PutQuote(quote); /* Write to the output file */
    }
}
### GETTING STARTED WITH THE COMPILER

This section highlights ways the documentation can help you understand how to run your program, use options to modify program compilation and execution, and debug your program.

```c
if (GetQuote(quote));    /* Read another quote */

exit(0);                  /* Everything worked */

static int GetQuote(char *quote)  /* get a quote */

fgets(quote, MAX_QUOTE_LENGTH, stdin);

if (feof(stdin)) {        /* Check for error */
    fputs("Error occurred while reading the quote file.

    stdev; exit(8);
}

return feof(stdin);

return 0;

static void Encrypt(char *quote)  /* Encrypt - encrypts a quote. A new encryption */

for (i = 0; i < 52; i++)
    for (j = i + 1; j < 52; j++)
        if (rand() % 2)
            alphabet[i] = alphabet[j];

for (j = 0; j < 52; j++)
    if (alphabet[i] != alphabet[j])
        alphabet[i] = alphabet[j];

int call = 0;

printf("%d");               /* Write the quote number */

puts(quote, stdout);        /* And the quote itself */

define("\n
    stdev; exit(8);

GetQuote - read a line from the quote file. If an error occurs */
end the program with a return code of 8. Return */
non-zero if at end-of-file, else return zero. */

static int GetQuote(char *quote) {
    /* Read a quote */
    fgets(quote, MAX_QUOTE_LENGTH, stdin);
    if (feof(stdin)) {
        /* Check for error */
        fputs("Error occurred while reading the quote file.

         stdev; exit(8);
    }
    return feof(stdin);
    return 0;
}

static void Encrypt(char *quote)  /* Encrypt - encrypts a quote. A new encryption */

for (i = 0; i < 52; i++)
    for (j = i + 1; j < 52; j++)
        if (rand() % 2)
            alphabet[i] = alphabet[j];

for (j = 0; j < 52; j++)
    if (alphabet[i] != alphabet[j])
        alphabet[i] = alphabet[j];

int call = 0;

printf("%d");               /* Write the quote number */

puts(quote, stdout);        /* And the quote itself */

define("\n
    stdev; exit(8);

PutQuote - writes an encrypted quote to the encryption file. */
if an error occurs, ends the process with a return */
non-zero if at end-of-file, else return zero. */

static void PutQuote(char *quote)  /* PutQuote - writes an encrypted quote to the encryption file. */

for (i = 0; i < 52; i++)
    for (j = i + 1; j < 52; j++)
        if (rand() % 2)
            alphabet[i] = alphabet[j];

for (j = 0; j < 52; j++)
    if (alphabet[i] != alphabet[j])
        alphabet[i] = alphabet[j];

int call = 0;

printf("%d");               /* Write the quote number */

puts(quote, stdout);        /* And the quote itself */

define("\n
    stdev; exit(8);

### Running Your Program

The compiler translates your program in source code form into machine-level code. The SAS/C Compiler does this in three phases. The first phase analyzes your program for correctness of C syntax and the second phase generates machine code from the output of phase 1. The third phase optimizes the machine code. The output from this process is called an object module.

After compilation, if you are working with two or more separately compiled programs, a link step is needed. Linking combines the separately compiled object modules into one executable file and ensures that any references between the separate object modules are resolved. The output from the link step is called a load module. (In addition to the IBM linkage editor, the SAS/C Compiler provides another link utility called CLINK, which is explained shortly.)

After these steps are completed, you can run your program. (By specifying certain options along the way, you can run your program using the SAS/C Debugger, more on this shortly.)

The SAS/C Compiler supplies options that allow you to modify each of these processes. There are compiler options that you can use during the compilation and linkage steps. There are run-time options that you can supply during the execution phase.

As noted earlier, the SAS/C Compiler and Library User’s Guide devotes a chapter covering each of these steps for each mode of execution (TSO, CMS, MVS-batch). Options and how to specify them are included in these chapters as well as in Chapters 8 and 9. The following sections trace these steps via the documentation for running the example program SQUOTE under TSO.

### Compiling, Linking, and Executing SQUOTE

Chapters 5, 6, and 7 are organized in a similar manner so that if you want to run your program under CMS or MVS-batch as well as under TSO you can find the appropriate reference sections quickly. The steps involved in running your program under each environment are described in these sections in the User’s Guide:

- **Compiling C Programs**
- **Using the Object Module Disassembler**
- **Linking C Programs**
- **Linking All-Resident Programs**
- **Running C Programs**

The SAS/C Compiler expects certain data sets to be present in order to produce and save appropriate files. These files are for:

- source code
- object code
- load module
- debugger.

Under TSO these can be allocated automatically by your site; if not, you will need to allocate them.

To specify a data set, you must follow standard TSO naming conventions. These are reviewed in Chapter 5. If you do not enclose the data set name in apostrophes, the CLISTS assume certain final qualifiers. For example, the LC370 CLIST assumes that the final
qualifier of your source data set name is "C." (If you don't specify it, "C" is automatically added. Note that this can be changed by your site.) For this reason, it is recommended that, under TSO, you name your data sets (all partitioned data sets, or PDSs) as follows:

- source data set - userid.pdsname.C
- object data set - userid.pdsname.OBJ
- load data set - userid.pdsname.LOAD
- debugger data set - userid.pdsname.DGBLIB

The example, SQUOTE, resides in a C source file called 'YOURID.PDSNAME.C.'

Compiling Your Program

Under TSO, the compiler and utility programs can be invoked using CLISTS. The syntax for invoking the CLIST supplied with the compiler, LC370, is

LC370 dsname <options>

To help you decide what options are appropriate, Table 8.1 in Chapter 8, "Compiler Options," (in the User's Guide) lists all options and their default values. If you want to use the defaults, the syntax for compiling our example SQUOTE is as follows:

LC370 PDSNAME(SQUOTE) PR("'PROJ.ePGMS.LIST'")

However, you may want to change some of these default values. For example, what if you want to output a formatted output listing to a specific list file and you don't want a cross-referencing listing produced at this point? Then you specify the following options (shown in their short form):

LC370 PDSNAME(SQUOTE) PR("'PROJ.CPGMS.LIST'") SD NOX

This means

- Invoke the CLIST to compile SQUOTE.
- Since OBJECT is the default, object code is stored by default in the data set YOURID.PDSNAME.OBJ, member SQUOTE.
- The PRINT option specifies that the listing file should be generated and stored in the data set PROJ.CPGMS.LIST.
- Since this is not one of your data sets, it is set off by three quotation marks.
- NOLIB is the default, but if you needed to include header files other than the standard ones, you would specify the LIB option here.
- SOURCE requests a formatted source listing.
- NOX indicates that you don't want a cross-reference list.

Using the Object Module Disassembler

The object module disassembler (OMD) is a useful debugging tool that provides a copy of the assembler code generated for a C program. If you specify the LINENO compiler option when using the OMD, your C source code is merged with the assembler instructions. You can choose to run the OMD with a compiler option, or you can use the OMD370 CLIST as follows:

OMD370 PDSNAME(SQUOTE)

You can supply various OMD options; these are covered in Chapter 8 of the User's Guide.

Linking Your Program

SAS/C provides an object code processor called CLINK. Each of Chapters 5, 6, and 7 in the User's Guide covers in detail when you must use the preprocessor before the linkage editor.

Under TSO, the CLK370 CLIST invokes the CLINK object code preprocessor, followed by the linkage editor. The CLINK preprocessor step can be skipped by specifying the NOCLINK option when you invoke the CLK370 CLIST. In general, however, it's advisable to invoke the CLK370 CLIST and allow both the preprocessor and linkage editor to process your object code. The syntax for this is

CLK370 PDSNAME(SQUOTE)

You can specify many keywords and options to control how your object code is processed and linked. These options are discussed and summarized in tables in the section "The CLK370 CLIST" in Chapter 5 of the User's Guide (and similar sections in Chapters 6 and 7). The output from this step is a load module that you can then execute.

Running Your Program

There are a number of ways to tell the operating system to execute a load module. Under TSO, your site has the option of letting you use the standard TSO CALL command or of providing higher level support. The TSO CALL command is

CALL PDSNAME(SQUOTE)

In the earlier section describing Chapter 9 in the User's Guide, it was pointed out that you can supply information to your program by passing run-time arguments or specifying run-time options. For example, if you want to see what release numbers are associated with the resident and transient libraries (the two parts of the run-time library) you can specify the =version option as follows:

CALL PDSNAME(SQUOTE) =version

In summary, under TSO, the steps for running your program consist of submitting a combination of the following CLISTS:

LC370  (optional)
CLK370  (optional)
TSO CALL command

Regardless of the environment (TSO, CMS, or MVS-batch), you can use the documentation for assistance in compiling and running your programs. First, select the chapter in the SAS/C Compiler and Library User's Guide (Chapter 5, 6, or 7) appropriate for your environment. Then use that chapter in conjunction with Chapter 8, Compiler Options," and Chapter 9, "Run-Time Argument Processing."

Running the Debugger

Whether you are working under TSO, CMS, or MVS-batch to run the SAS/C Debugger, you usually perform the following tasks:

- compile using the DEBUG option
- run CLINK (optional)
- request DEBUG at run time.

Since the example program SQUOTE is being run under TSO, the debugger library must also be allocated for use during the debug session. The syntax for the allocation for our example is
Since the data sets for SQUOTE are named as shown earlier, the syntax for each step involved in using the debugger follows.

1. \texttt{LC370 PDSNAME(SQUOTE) DEBUG}
2. \texttt{CLK370 PDSNAME(SQUOTE)}
3. \texttt{CALL PDSNAME(SQUOTE) =d}

Step 1 compiles SQUOTE and places the output in \texttt{YOURID.PDSNAME.OBJ} with the member being a section name (portion of your program) assigned by the compiler by default or by specifying the \texttt{SQUOTE} compiler option.

Step 2 creates a load module of SQUOTE and puts it in \texttt{YOURID.PDSNAME.LOAD} with the member name SQUOTE. (Again, you can bypass CLINK, if possible for your application.)

Step 3 runs SQUOTE with \texttt{=d} invoking the debugger.

For complete information on using the SAS/C Debugger, and helpful pointers for debugging your programs, you can again refer to the \textit{SAS/C Source Level Debugger User's Guide}.

\section*{Pointers to Efficiency Tips}

There are several sections in the four basic books where the emphasis is on helping you use the SAS/C Compiler and Library efficiently. Once you become familiar with the material in these sections, you can save time and resources in running your applications. Good places to start looking for this type of information include the following:

- "Efficient Programming with the SAS/C Compiler" (User's Guide, Chapter 4)
- "Using in-line functions to generate optimized code" (User's Guide, Chapter 4)
- "The OPTIMIZE Option" (User's Guide, Chapter 4)
- "Efficient I/O" (Library Reference, Vol. 1, Chapter 12)

\subsection*{Efficiency and the SQUOTE Example}

As an example of how the documentation can continue to help you fine-tune your programs, this section takes one of the processes covered in the material above, in-line functions, and uses it to illustrate how the SQUOTE program can execute more efficiently. The SAS/C documentation provides both the conceptual background and technical details to enable you to understand and use this methodology, associated options, and related material. Even if you feel the majority of your applications are not "complex" enough to merit worrying about optimization and efficiency, the results from reading and experimenting may be surprisingly successful.

\subsection*{Using In-Line Code}

One way to make programs more efficient is to replace a call to a function with the function itself. This way, the overhead associated with building a parameter list and calling the function is eliminated from the function doing the calling. An \textit{in-line function} is a function for which the compiler replaces a call to the function with the code for the function itself. This process of replacing a function call with the function's code is called \textit{inlining}.

The \texttt{OPTIMIZE} compiler option is used to enable global optimization of your program's code. This means that the flow of control and data through functions is made more efficient. One of the benefits of using this option is that it makes the \texttt{INLINE} option active by default.

Many of the SAS/C functions are declared in \texttt{<stdio.h>} with the \_\texttt{inline} run-time keyword as the default. The options \texttt{error()} and \texttt{eof()} used in SQUOTE are such options. Using the \texttt{OPTIMIZE} option causes the body of both functions to replace the function calls in our example.

If you use the \texttt{INLOCAL} option in addition to the \texttt{OPTIMIZE} option, all single-call static functions are inlined. This means both the \texttt{Encrypt} and \texttt{PutQuote} functions are inlined because they are static functions that are only called once in the source file.

Going one step further, you could also specify the \texttt{COMPLEXITY} option. This option provides another way to use inlining without having to specify the \_\texttt{inline} keyword (as was done in the header files earlier). By default, the complexity of a function is 0. But, if you specify a complexity greater than 0 (such as 1) and the \texttt{INLINE} option is also in effect (as it is here), small functions are inlined automatically. If you want to inline more complex functions, then a higher value can be provided (covered in "Compiler Options for Inlining" in Chapter 4 of the User's Guide) to force additional functions to be inlined. If this is done for our SQUOTE example by specifying \texttt{COMPLEXITY}(5), then the \texttt{GetQuote} function is inlined as well.

This is an example of one of several efficiency tips covered in the documentation. Additional suggestions are noted in other sections as well as in other chapters throughout the book.

\section*{Conclusions and Where to Go From Here}

This paper has provided examples of how the documentation accompanying the SAS/C Compiler can help you efficiently write and run programs using the SAS/C implementation of the C language. For each of the following areas, the documentation provides both conceptual and reference material to assist you in your program development:

- writing C programs
- executing C programs
- debugging C programs
- producing efficient C programs

From here you may want to investigate some of the features covered in Chapters 10-14 and Appendices 3-6 of the SAS/C Compiler and Library User's Guide. There are also additional sections in Volume 1 of SAS/C Library Reference covering signal-handling and multibyte character functions, for example. For specialized applications such as inter-user communication (IUCV) you'll want to consult Volume 2 of SAS/C Library Reference.

In addition, other SAS/C documentation is available, as referenced in the introduction to this paper. If you are interested in writing programs that use C and one or more other high-level languages at the same time, such as FORTRAN or Pascal, then you can refer to the SAS/C Compiler Interlanguage Communication Feature User's Guide. The SAS/C Full-Screen Support Library User's Guide accompanies the SAS/C Full-Screen Support Library which is available as a separate product. This documentation also includes a tutorial and shows you how you can develop C programs for a wide range of
full-screen applications. To provide an entry point for all of this, SAS Technical Report C-108, Master Index to SAS/C Compiler Documentation is now available. Additional technical reports are available as noted earlier.

This completes a "hands on" tour of the basic documentation set accompanying the SAS/C Compiler and Library. Whether you use the documentation to learn about features of SAS/C or as a quick reference, or as both, familiarity with the contents will help you produce more efficient and effective programs.

ADDITIONAL SOURCES OF INFORMATION


ACKNOWLEDGEMENTS

Thanks to the following persons who helped make this paper and the reference aids to the SAS/C documentation possible.

- Tim Hunter (Manager, IBM/370 C Compilers) for his assistance with the example for this paper.

- Jodie Gilmore and Harriet Watts (Documentation Development) for their attention to reference details.

- Editors responsible for indexing the documentation.

SAS/C is a registered trademark of SAS Institute Inc., Cary, NC, USA.

MVS/XA and MVS/ESA are trademarks of International Business Machines Corporation.