SAS/C Software: Changes and Enhancements

Release 6.50

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/C Software: Changes and Enhancements, Release 6.50*, Cary, NC: SAS Institute Inc., 1997. pp. 109.

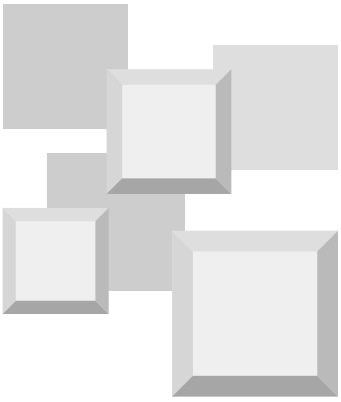
SAS/C Software: Changes and Enhancements, Release 6.50

ISBN 1–58025–135–8

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

1st printing, March 1998

Contents



PART 1

Changes and Enhancements

CHAPTER

1

SAS/C Compiler Changes in Release 6.50

Introduction

This chapter describes the changes and enhancements to the SAS/C Compiler for Release 6.50.

Release 6.50 Compiler Enhancements

The following changes and enhancements to the SAS/C Compiler have been implemented with Release 6.50:

- Compiler Definition Change
- New Compiler Options
- Language Extensions
- #chain Command

Compiler Definition Change

Until Release 6.50, the SAS/C compiler defined the symbol **`I370`** to indicate compilation was being performed for the IBM System/370 mainframe. However, the **`I370`** symbol violates the user's name space defined by ANSI standards. The Release 6.50 SAS/C Compiler now defines a new symbol, **`__I370__`**, to conform to ANSI standards.

Note: This fix has also been made to Release 6.00.03; so any cross compiler after 6.00.02 will also define **`__I370__`**. △

The Release 6.50 compiler will continue to define the **`I370`** symbol, but it is *not* recommended for use.

Note: This change has no affect on the C++ translator. △

New Compiler Options

The SAS/C Compiler accepts a number of options enabling you to alter the way code is generated, the way listing files appear, as well as other aspects of the compilation. This section describes the new compiler options available with Release 6.50 and how they are implemented in various IBM mainframe environments (CMS, TSO, and MVS Batch).

New Options Summary

Table 1.1 on page 4 summarizes all the new compiler options for Release 6.50. This table is an extension of "Table 6.1 Compiler Options" in the *SAS/C Compiler and Library User's Guide*.

The first column lists the new digraph options in long form for the IBM 370. Capital letters indicate the abbreviation for the option. The second column indicates the default for each option. For the default value of the digraph option, you are referred to the description of the option later in the chapter. The third column indicates how the option is specified from the OpenEdition shell. The fourth column indicates whether the option can be negated. An exclamation point (!) means the option can be negated, and a plus sign (+) means it can not. The description of the digraph option identifies the negated form of the option. The next three columns represent the environment(s) for which an option is implemented. An asterisk (*) indicates the option affects this environment. The Affects Process column names the process that is affected by the option. The C in the Affects Process column indicates that compilation is affected by the option. An asterisk in the Sys column warns that the form or meaning of the option may differ depending on the environment in which the compiler is running.

Note: Under MVS batch, the OpenEdition shell, and CMS, if you specify contradictory options, the option specified last is used. △

Table 1.1 New Options

Option Name	Default	OpenEdition	Negation	MVS Batch	TSO	CMS	Affects Process	Sys
AUtoinst	**NOAUtoinst**	**-Kautoinst**	!	*	*	*	C	
DBGObj	**NODBGObj**	**-Kdbgobj**	!	*	*	*	C	
DIgraph	see description	**-Kdigraph[n]**	!	*	*	*	C	

New Options

autoinst (**-Kautoinst** under OpenEdition)

The **autoinst** option controls automatic implicit instantiation of template functions and static data members of template classes. The compiler organizes the output object module so that COOL can arrange for only one copy of each template item to be included in the final program. In order to correctly perform the instantiation, the **autoinst** option must be enabled on a compilation unit that contains both a use of the item and its corresponding template identifier. (See the *SAS/C C++ Development System User's Guide, Second Edition, Release 6.50* for information about templates and automatic instantiation.)

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

dbgobj (**-Kdbgobj** under OpenEditon)

The **dbgobj** option causes the compiler to place the debugging information in the output object file, instead of a separate debugger file. If the debugging information is not placed in the object file, you cannot debug the automatically instantiated objects.

If automatic instantiation is specified with the **autoinst** option, **dbgobj** is enabled automatically.

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

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

digraph (**-Kdigraph** under OpenEdition)

Digraph options enable the translation of the International Standard Organization (ISO) digraphs and/or the SAS/C digraph extensions.

A digraph is a two character representation for a character that may not be available in all environments. The different options allow you to enable subsets of the full diagraph support offered collectively by ISO and SAS/C. Table 1.2 on page 4 gives a brief description of the new digraph compiler options.

Table 1.2 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 1.3 on page 4 provides the default values and an example of how to negate the options in each of the different environments.

Table 1.3 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 OpenEdition	**-Kdigraph1**, **-Kdigraph3**	**!Kdigraph1**, **!Kdigraph3**

Table 1.4 on page 5 lists several of the ISO digraph sequences from the C++ ANSI draft. Basically, the alternative sequence of characters is an alternative spelling for the primary sequence. Similar to SAS/C digraphs, substitute sequences are not replaced in either string constants or character constants. SAS/C Release 6.50 currently supports the left column of pairs of primary and alternative sequences.

Table 1.4 ISO digraph Alternative Tokens

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

Note: See Chapter 2, "Special Character Support", in the *SAS/C Compiler and Library User's Guide* for more information on digraphs. △

Compiler Option Clarification

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 "Compiling C Programs" in the *SAS/C Compiler and Library User's Guide*. △

New External Compiler Variables

Older versions of MVS 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 MVS (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 MVS 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. △

New Compiler Comment Support

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

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

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

Extended @ Operator Capability

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

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

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

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

New Character and String Qualifiers

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

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

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

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

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

#chain Command

#chain *FILENAME*

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

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

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

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

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

#chain //DDN:PROJECT(MAPPINGS)

alice.h betty.h
```

If the DDN "PROJECT" was defined so that MAPPINGS contained

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

Then the header map list would be :

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

CHAPTER

2

SAS/C OpenEdition Library Changes in Release 6.50

Introduction

This chapter describes the changes and enhancements to the OpenEdition SAS/C Library Functions for Release 6.50.

Release 6.50 Enhancements

The following OpenEdition SAS/C Library Functions have been added for Release 6.50:

`chpriority` - Change process priority

`endgrent` - Close the group database

`endpwent` - Close the user database

`extlink` - Define an external link

`getgrent` - Access group database sequentially

`getitimer` - Obtain interval timer values

`getpgid` - Get process group id for a process

`getpriority` - Determine process priority

`getpwent` - Access user database sequentially

`getrlimit` - Obtain OpenEdition resource limits

`getrusage` - Obtain OpenEdition usage statistics

`getsid` - Get session leader id for a process

`getwd` - Get working directory path name

`lchown` - Change file or link ownership

`mmap` - Map an HFS file into memory

`mprotect` - Change memory protection mapped to file

msgctl - Control a message queue

msgget - Create or find a message queue

msgrcv - Receive a message from a message queue

msgsnd - Send a message to a message queue

msgxrcv - Receive message with sender information

msync - Synchronize a memory mapped HFS file

munmap - Cancel mapping of a memory area to a file

oetaskctl - Control subtasks with OpenEdition

readextlink - Read an external link

realpath - Return absolute pathname

semctl - Control a semaphore set

semget - Create or find a set of semaphores

semop - Update semaphores atomically

setgrent - Reposition the group database

setitimer - Define an interval timer

setpriority - Change process priority

setpwent - Reposition the user database

setregid - Set real and/or effective group ID

setreuid - Set real and/or effective user ID

setrlimit - Define OpenEdition resource limits

shmat - Attach a shared memory segment

shmctl - Control a shared memory segment

shmdt - Detach a shared memory segment

shmget - Create or find a shared memory segment

sigblkjmp - Intercept longjmp without signal mask change

spawn - Spawn a New Process

spawnp - Spawn a New Process

tcgetsid - Get session leader id for a process

truncate - Truncate an HFS file

New OpenEdition Library Functions

This section describes the new library functions available to Release 6.50 SAS/C users under OpenEdition.

chpriority

Change process priority

SYNOPSIS

```
#include <sys/resource.h>
int chpriority(int kind, id_t id,
               int form, int prio);
```

DESCRIPTION

The **chpriority** function changes the OpenEdition priority of a process, or of all processes in a process group or belonging to a user. See the **getpriority** function description in section "getpriority" on page 16 for further information on OpenEdition priorities.

The **kind** argument to **chpriority** should be specified as a symbolic constant indicating the scope of the priority change. The permissible values are:

- **PRIO_PROCESS** - Specifies that the **id** argument is the pid of the process whose priority is to be changed.
- **PRIO_PGRP** - Specifies that the **id** argument is the pid of the process group whose processes should be changed in priority.
- **PRIO_USER** - Specifies that the **id** argument is the uid of the user whose processes are to be changed in priority.

The **id** argument specifies the process id, process group id, or user id whose priority should be changed. If **id** is 0, the calling process id, process group id, or user id is specified.

The **form** argument specifies whether the **prio** argument is an absolute or relative priority. It should be specified as one of the following symbolic constants:

- **CPRIO_ABSOLUTE** - Specifies that **prio** is to be the new priority of the processes specified.
- **CPRIO_RELATIVE** - Specifies that **prio** is the amount which should be added to the existing priority of each process specified.

The **prio** argument specifies the requested new priority, or the amount by which the priority should be changed, depending of the value of the **form** argument. Priorities are restricted to the range of -20 to 19, where lower numbers indicate higher priority.

Note: OpenEdition sites must enable the use of the **chpriority** function. If the use of **chpriority** has not been enabled, any use of **chpriority** will fail with errno set to **ENOSYS**. △

RETURN VALUE

chpriority returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **chpriority** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE

This example is compiled using **sascc370 -Krent -o**. This program demonstrates the use of the Open Edition process priority functions **chpriority()**, **getpriority()**, and **setpriority()**.

```
/*---------------------------+
| POSIX/UNIX header files    |
+---------------------------*/
#include <unistd.h>

#include <sys/types.h>
```

```
#include <errno.h>

/*---------------------------+
| ISO/ANSI header files      |
+---------------------------*/
#include <stdlib.h>

#include <stdio.h>

#include <errno.h>
/*---------------------------+
| Name:     main             |
| Returns: exit(EXIT_SUCCESS)|
| or exit(EXIT_FAILURE)      |
+---------------------------|
+---------------------------*/
int main()
{
/* generic return code      */
   int rc;

/* which kind of process    */
/* id to use                */
   int kind;

 /* process id              */
   id_t id;

/* is oeprty relative or    */
/* absolute                 */
   int form;

/* process priority         */
/* (version 1)              */
   int prty1;

/* process priority         */
/* (version 2)              */
   int prty2;

/* process id from getpid() */
   pid_t pid;

/* process group id from     */
/* getpgid()                 */
   pid_t pgid;

/* process user id from      */
*/ getuid()                  */
   uid_t uid;

/* get the user id for this   */
/* process                    */
   uid = getuid();

/* get the process id for     */
/* this process               */
   pid = getpid();
/* get the group process id   */
/* for this process  */
   pgid = getpgid(pid);

   if (pgid == -1)
   {
    perror("Call to getpgid failed");
    exit(EXIT_FAILURE);
   }

   printf("      The process id: %d\n",
     (int)pid);
   printf("The process group id: %d\n",
     (int)pgid);
   printf(" The process user id: %d\n",
     (int)uid);

   /*-----------------------------*/
   /* Get the process priority     */
   /* using the process id         */
   /*-----------------------------*/
   print
     ("\nGet the Process Priority
         using the Process ID\n");

/* the id arg is the pid of a process */

   kind = PRIO_PROCESS;

   /* version 1 */
   id = (id_t)pid;

/* Set errno to zero for    */
/* error handling           */
   errno = 0;
   prty1 = getpriority(kind, id);

/* ----------------------------------*/
/* Test for Error                    */
/* Note:                             */
/* getpriority() may return a '-1'  */
/* return code for either a          */
/* failure rc, or when the priority */
/* is in-fact '-1'.  To distinguish */
/* between the two conditions,       */
/* check the errno                   */
/* value for a non-zero value.       */
/*----------------------------------*/
 if (prty1 == -1 && errno != 0)
 {
   perror("Call to getpriority failed");
   exit(EXIT_FAILURE);
   }
   else
   {
      printf("The process priority
       (pid:version 1): %d\n", prty1);
   }

/* version 2 */
/* 0 implies current processs id  */
   id = (id_t)0;

/* Reset errno to zero for        */
```

```
/* error handling                 */
   errno = 0;
   prty2 = getpriority(kind, id);

/* Test for Error */
   if (prty2 == -1 && errno != 0)
   {
      perror("Call to getpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
     printf("The process priority
     (pid:version 2): %d\n", prty2);
   }

/*-----------------------------*/
/* Get the process priority    */
/* using the group process id  */
/*-----------------------------*/
   printf("\nGet the Process Priority
    using the Group Process ID\n");

/* the id arg is the group process id */
   kind = PRIO_PGRP;

/* version 1 */
   id = (id_t)pgid;

/* Set errno to zero for error handling      */
   errno = 0;
   prty1 = getpriority(kind, id);

/* Test for Error */
   if (prty1 == -1 && errno != 0)
   {
      perror("Call to getpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      printf("The process priority
       (gpid:version 1): %d\n", prty1);
   }

/* version 2 */
/* 0 implies current group processs id       */
   id = (id_t)0;

/* Reset errno to zero for error handling    */
   errno = 0;
   prty2 = getpriority(kind, id);

/* Test for Error */
   if (prty2 == -1 && errno != 0)
   {
      perror("Call to getpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      printf("The process priority
       (gpid:version 2): %d\n", prty2);
   }

 /*-----------------------------------------*/
 /* Get the process priority using the      */
 /* process User id                         */
 /*-----------------------------------------*/
 print
  ("\nGet the Process Priority of the User ID\n");

/* the id arg is the user id of the process  */
   kind = PRIO_USER;

/* version 1 */
   id = (id_t)uid;
/* Set errno to zero for error handling      */
   errno = 0;
   prty1 = getpriority(kind, id);

   /* Test for Error */
   if (prty1 == -1 && errno != 0)
   {
      perror("Call to getpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      printf("The process priority (uid:version 2):
        %d\n", prty1);
   }

/* version 2 */
/* Reset errno to zero for error handling    */
   errno = 0;

 /* 0 implies current process user id        */
   id = (id_t)0;
   prty2 = getpriority(kind, id);

   /* Test for Error */
   if (prty2 == -1 && errno != 0)
   {
      perror("Call to getpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      printf("The process priority (uid:version 2):
       %d\n", prty2);
   }

/*-----------------------------------------------*/
/* Set the process priority using the            */
/* process id                                    */
 /*-----------------------------------------------*/
 printf
("\nSet the Process Priority using
   the Process ID\n");

/* the id arg is the pid of a process        */
```

```
   kind = PRIO_PROCESS;

/* an id of 0 implies current processs id    */
   id = (id_t)0;

/* Reset errno to zero for error handling    */
   errno = 0;

/* Set process priority to 5                 */
   prty1 = 5;

   rc = setpriority(kind, id, prty1);

   /*-------------------------------------------*/
   /* Test for Error                            */
   /* Note: OpenEdition sites must enable the use */
   /*       of the setpriority() function. If the */
   /*       use of setpriority() has not been     */
   /*       enabled, any use of setpriority()     */
   /*       will fail with errno set to ENOSYS.   */
   /*                                             */
   /*-------------------------------------------*/
   if (rc == -1)
   {
      perror("Call to setpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);

      /* Test for Error */
      if (errno != 0)
      {
         perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
      }
      printf("The process priority is now (pid):
        %d\n", prty2);
   }

   /*-------------------------------------------*/
   /* Set the process priority using the group   */
   /* process id                                */
   /*-------------------------------------------*/
 printf
("\nSet the Process Priority using
   the Group Process ID\n");

/* the id arg is the group id of the process */
   kind = PRIO_PGRP;

/* 0 implies current group processs id       */
   id = (id_t)0;

/* Reset errno to zero for error handling    */
   errno = 0;

/* Set process priority to 10                */
   prty1 = 10;

   rc = setpriority(kind, id, prty1);

   /* Test for Error */
   if (rc == -1)
   {
      perror("Call to setpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);

      /* Test for Error */
      if (errno != 0)
      {
         perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
      }
      printf("The process priority is now (gpid):
       %d\n", prty2);
   }

 /*-------------------------------------------*/
 /* Set the process priority using the        */
 /*process User id                            */
 /*-------------------------------------------*/
  printf
  ("\nSet the Process Priority of
     the User ID\n");

/* the id arg is the user id of the process  */
   kind = PRIO_USER;

/* an id of 0 implies current user id        */
   id = (id_t)0;

/* Reset errno to zero for error handling    */
   errno = 0;

/* Set process priority to 15                */
   prty1 = 15;


   rc = setpriority(kind, id, prty1);

/* Test for Error */
   if (rc == -1)
   {
      perror("Call to setpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);

      /* Test for Error */
      if (errno != 0)
      {
         perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
```

```
      }
printf
("The process priority is now
   (uid): %d\n", prty2);
   }

/*-------------------------------------------*/
/* Change the process priority using the     */
/* process id                                */
/*-------------------------------------------*/
printf
 ("\nChange the Process Priority
    using the Process ID\n");

/* the id arg is the pid of a process        */
   kind = PRIO_PROCESS;

/* an id of 0 implies current processs id    */
   id = (id_t)0;

/* Reset errno to zero for error handling    */
   errno = 0;

/* change using "absolute"                   */
/* priority - equivalent to setpriority()    */
   form = CPRIO_ABSOLUTE;
   printf("\tChange using CPRIO_ABSOLUTE\n");

/* Change process priority to 3              */
   prty1 = 3;

   rc = chpriority(kind, id, form, prty1);

/*-------------------------------------------------*/
/* Test for Error                                  */
/* Note: OpenEdition sites must enable the use     */
/*       of the chpriority() function.             */
/*       If the use of chpriority() has not been   */
/*       enabled, any use of chpriority() will     */
/*       fail with errno set to ENOSYS.            */
/*                                                 */
/*-------------------------------------------------*/
   if (rc == -1)
   {
      perror("Call to chpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);

      /* Test for Error */
      if (errno != 0)
      {
         perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
      }
  printf
  ("The process priority is now
     (pid): %d\n", prty2);
   }

/* change using "relative" priority */
   form = CPRIO_RELATIVE;
   printf("\tChange using CPRIO_RELATIVE\n");

/* Bump process priority up by 2             */
   prty1 = 2;

   rc = chpriority(kind, id, form, prty1);

   /* Test for Error */
   if (rc == -1)
   {
      perror("Call to chpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);

      /* Test for Error */
      if (errno != 0)
      {
       perror("Call to getpriority failed");
       exit(EXIT_FAILURE);
      }
   printf("The process priority is now (pid):
     %d\n", prty2);
   }

/*-------------------------------------------*/
/* Change the process priority using the     */
/* group process id                          */
/*-------------------------------------------*/
 printf
  ("\nChange the Process Priority using
     the Group Process ID\n");

/* the id arg is the group id of the process */
   kind = PRIO_PGRP;

/* 0 implies current group processs id       */
    id = (id_t)0;

/* Reset errno to zero for error handling    */
   errno = 0;

   /* change using "absolute"                */
   /* priority - equivalent to setpriority()  */
   form = CPRIO_ABSOLUTE;
   printf("\tChange using CPRIO_ABSOLUTE\n");

 /* Change process priority to 7              */
   prty1 = 7;

   rc = chpriority(kind, id, form, prty1);

   /* Test for Error */
   if (rc == -1)
   {
      perror("Call to chpriority failed");
```

```
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);

      /* Test for Error */
      if (errno != 0)
      {
         perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
      }
  printf
("The process priority is now
    (gpid): %d\n", prty2);
   }

/* change using "relative" priority */
   form = CPRIO_RELATIVE;
   printf("\tChange using CPRIO_RELATIVE\n");

/* Bump process priority up by 3            */
   prty1 = 3;

   rc = chpriority(kind, id, form, prty1);

   /* Test for Error */
   if (rc == -1)
   {
      perror("Call to chpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);

      /* Test for Error */
      if (errno != 0)
      {
         perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
      }
 printf("The process priority is now (gpid):
   %d\n", prty2);
   }

/*--------------------------------------*/
/* Change the process priority using    */
/* the process User id                  */
/*--------------------------------------*/
printf
("\nChange the Process Priority
   of the User ID\n");

/* the id arg is the user id of the process  */
   kind = PRIO_USER;

/* 0 implies current group processs id       */
   id = (id_t)0;

/* Reset errno to zero for error handling    */
   errno = 0;

/* change using "absolute"                   */
/*  priority - equivalent to setpriority()  */
   form = CPRIO_ABSOLUTE;
   printf("\tChange using CPRIO_ABSOLUTE\n");

/* Change process priority to 11           */
   prty1 = 11;

   rc = chpriority(kind, id, form, prty1);

   /* Test for Error */
   if (rc == -1)
   {
      perror("Call to chpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);

      /* Test for Error */
      if (errno != 0)
      {
       perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
      }

printf("The process priority is now (uid):
   %d\n", prty2);
   }

/* change using "relative" priority */
   form = CPRIO_RELATIVE;
   printf("\tChange using CPRIO_RELATIVE\n");

/* Bump process priority up by 4            */
   prty1 = 4;

   rc = chpriority(kind, id, form, prty1);

   /* Test for Error */
   if (rc == -1)
   {
      perror("Call to chpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);

      /* Test for Error */
      if (errno != 0)
      {
      perror("Call to getpriority failed");
       exit(EXIT_FAILURE);
      }
      printf("The process priority is now (uid):
         %d\n", prty2);
   }
```

```
    exit(EXIT_SUCCESS);

}  /* end of main() */
```

RELATED FUNCTIONS
getpriority, **setpriority**

endgrent

Close the group database

SYNOPSIS

```
#include <sys/types.h>
#include <grp.h>
void endgrent(void);
```

DESCRIPTION
The **endgrent** function closes the OpenEdition group database once the **getgrent** function is complete. If **getgrent** is called after a call to **endgrent**, the database will be opened once again, and information about the first group returned.

RETURN VALUE
endgrent has no return value.

USAGE NOTES
The **endgrent** function can only be used with MVS 5.2.2 or a later release.

If the **endgrent** function cannot be executed (for instance, because OpenEdition is not installed or running), it issues an MVS user **ABEND 1230** to indicate the error.

RELATED FUNCTIONS
getgrent, **setgrent**

endpwent

Close the user database

SYNOPSIS

```
#include <sys/types.h>
#include <pwd.h>
void endpwent(void);
```

DESCRIPTION
The **endpwent** function closes the OpenEdition user database once the **getpwent** function is complete. If **getpwent** is called after a call to **endpwent**, the database will be opened once again, and information about the first user returned.

RETURN VALUE
endpwent has no return value.

USAGE NOTES
The **endpwent** function can only be used with MVS 5.2.2 or a later release.

If the **endpwent** function cannot be executed (for instance, because OpenEdition is not installed or running), it issues an MVS user **ABEND 1230** to indicate the error.

RELATED FUNCTIONS
getpwent, **setpwent**

extlink

Define an external link

SYNOPSIS

```
#include <unistd.h>
int extlink(const char *ename,
            const char *lname);
```

DESCRIPTION
The **extlink** functions creates a link from the OpenEdition hierachical file system to an external (non-HFS) file. The argument **ename** is the name of the external file, and **lname** specifies the name of the link. **lname** must specify the name of an HFS file. For programs not compiled with the

```
posix
```

option, style prefixes may be required. The external name, **ename**, must not specify a style prefix. See "File Naming Conventions" in the *SAS/C Library Reference, Volume 1*, for further information.

Note: External links are not transparent, similar to other OpenEdition symbolic links. That is, opening or unlinking an external link will not open or delete the referenced external file. External links are used by IBM's NFS implementation and can be used by other applications, but are not handled automatically by either OpenEdition or the SAS/C library. The contents of an external link can be accessed using the **readextlink** function. △

Note: An external link resembles a symbolic link, and that **S_ISLNK(s->st_mode)** will be true when **s** is the value stored by the **lstat** function for an external link. You can distinguish external links from other links using **S_ISEXTL(s->st_mode, s->st_genvalue)**, which will be non-zero only for an external link. △

RETURN VALUE
extlink returns **0** if successful, or **-1** if unsuccessful.

RELATED FUNCTIONS
lstat, **readextlink**

getgrent

Access group database sequentially

SYNOPSIS

```
#include <sys/types.h>
#include <grp.h>
struct group *getgrent(void);
```

DESCRIPTION

The **getgrent** function is used to read the system database that defines all OpenEdition groups. The first call to **getgrent** returns the first defined group, and each successive call returns information about the next group in the database. After information about the last group is obtained, **getgrent** returns **0** to indicate the end of file. When **0** is returned, the file position is reset, so that the next call to **getgrent** will retrieve the first database entry.

Note: It is more efficient to use the **getgrgid** or **getgrnam** function to obtain information about a specific group, rather than a loop calling **getgrent**. **getgrent** is provided mainly as an aid to porting existing UNIX programs that use it. △

RETURN VALUE

getgrent returns a pointer to a **struct group** if successful, and the end of the group database has not been reached. **getgrent** returns **0** if the end of the database has been reached, or if an error condition occurred. See the function description for **getgrgid** in the " **posix** Function Reference" section of the *SAS/C Library Reference, Volume 2* for more information about the definition of **struct group**.

Note: The pointer returned by **getgrent** may be a static data area that can be rewritten by the next call to **getgrent**, **getgrgid** or **getgrnam**. △

USAGE NOTES

The **getgrent** function can only be used with MVS 5.2.2 or a later release.

RELATED FUNCTIONS

endgrent, **getgrgid**, **getgrnam**, **setgrent**

getitimer

Obtain interval timer values

SYNOPSIS

```
#include <sys/time.h>
int getitimer(int kind,
              struct itimerval *val);
```

DESCRIPTION

The **getitimer** function returns information about an active interval timer. (See the **setitimer** function in section "setitimer" on page 41 for more information about interval timers.)

The **kind** argument is a symbolic constant that specifies the type of time interval for which information is wanted. The permitted values are:

- **ITIMER_REAL** – Specifies a real time interval. Each time the interval expires, a SIGALRM signal is generated.
- **ITIMER_VIRTUAL** – Specifies a virtual (CPU) time interval. Each time the interval expires, a SIGVTALRM signal is generated.
- **ITIMER_PROF** – Specifies a profiling interval, measuring CPU time plus system time used in behalf of this process. Each time the interval expires, a SIGPROF signal is generated.

The **val** argument is a pointer to an **itimerval** structure in which information about the current interval timer should be stored. See the **setitimer** function description for more information on the contents of a **struct itimerval**.

RETURN VALUE

getitimer returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **getitimer** function can only be used with MVS 5.2.2 or a later release.

RELATED FUNCTIONS

setitimer

getpgid

Get Process Group Id for a Process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgid(pid_t pid);
```

DESCRIPTION

The **getpgid** function returns the process group id for a specific process id. The argument **pid** should be the id of the process, or **0** to request the process group id for the current process.

RETURN VALUE

getpgid returns the process group id, or **-1** if it fails.

USAGE NOTES

The **getpgid** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE

This example is compiled using **sascc370 -Krent -o**.

```
/*-------------------------------------+
| POSIX/UNIX header files              |
+-------------------------------------*/
#include <unistd.h>

#include <sys/types.h>

/*-------------------------------------+
| ISO/ANSI header files                |
+-------------------------------------*/
#include <stdlib.h>

#include <stdio.h>

#include <errno.h>
/*-------------------------------------+
| Name:      main                      |
```

```
| Returns:   exit(EXIT_SUCCESS) or
| exit(EXIT_FAILURE)                  |
+-------------------------------------*
int main()
{
/* current process id from getpid()     */
   pid_t pid;

/* process group id from getpgid()      */
   pid_t pgid;

/*-----------------------------------------*/
/* Get the group process id of the current  */
/* process                                  */
/* Note: Both version 1 and version 2 are   */
/*       equivalent to calling the          */
/*.      getpgrp() function                 */
/*-----------------------------------------*/
printf
  ("\nGet the Group Process ID of the
    Current Process\n");

/* version 1                                 */
/* Set errno to zero for error handling      */
   errno = 0;

/* get the process id for this process       */
   pid = getpid();

/* get the group process id for this process  */
   pgid = getpgid(pid);

/* Test for Error */
   if (pgid == -1)
   {
      perror("Call to getpgid failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      printf
("      The process id: %d\n", (int)pid);
      printf("The process group id:%d\n",
        (int)pgid);
   }

/* version 2                                 */
/* Reset errno to zero for error handling    */
   errno = 0;

/* 0 implies current processs id             */
   pid = 0;

/* get the group process id for this process  */
   pgid = getpgid(pid);

/* Test for Error */
   if (pgid == -1)
   {
      perror("Call to getpgid failed");
      exit(EXIT_FAILURE);
   }
   else
   {
  printf
   ("The process group id: %d\n", (int)pgid);
   }

   exit(EXIT_SUCCESS);

}  /* end of main() */
```

RELATED FUNCTIONS

getpgrp, **setpgid**

getpriority

Determine process priority

SYNOPSIS

```
#include <sys/resource.h>
int getpriority(int kind, int id);
```

DESCRIPTION

The **getpriority** function obtains the OpenEdition priority of a process, a process group or a user. The priority is an integer between -20 and 19 which is used in scheduling process execution. Lower priority numbers are considered more urgent. These priority numbers are translated by OpenEdition in a site-specific manner into MVS SRM (system resources manager) specifications that control the priority of both OpenEdition and non-OpenEdition MVS processing. See the *IBM Publication OpenEdition MVS Planning* (SC23-3015) for more information on OpenEdition priorities and their interpretation.

The **kind** argument to **getpriority** should be specified as a symbolic constant indicating what kind of priority information is needed. The permissible values are:

- **PRIO_PROCESS** – Specifies that the **id** argument is the pid of the process whose priority is wanted.
- **PRIO_PGRP** – Specifies that the **id** argument is the pid of the process group whose priority is wanted.
- **PRIO_USER** – Specifies that the **id** argument is the uid of the user whose priority is wanted.

The **id** argument specifies the process id, process group id, or user id whose priority is needed. If **id** is **0**, the calling process id, process group id, or user is indicated.

If there is more than one process running that matches the arguments, for instance, multiple processes for a specified user, the smallest priority value for any process is returned.

RETURN VALUE

getpriority returns the requested priority if successful, or **-1** if unsuccessful. Since **-1** can also be returned as a priority value, you should set **errno** to **0** before calling **getpriority**, and test it for a non-zero value after the call to determine whether an error occurred.

USAGE NOTES

The **getpriority** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE

Refer to "chpriority" on page 8 for an example that demonstrates the use of the OpenEdition process priority functions **chpriority**, **getpriority**, and **setpriority**.

RELATED FUNCTIONS

chpriority, **setpriority**

getpwent

Access user database sequentially

SYNOPSIS

```
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwent(void);
```

DESCRIPTION

The **getpwent** function is used to read the system database defining all OpenEdition users. The first call to **getpwent** returns the first defined user, and each successive call returns information about the next user in the database. After information about the last user is obtained, **getpwent** returns **0** to indicate the end of file. When **0** is returned, the file position is reset, so that the next call to **getpwent** will retrieve the first database entry.

Note: It is more efficient to use the **getpwuid** or **getpwnam** function to obtain information about a specific user, rather than a loop calling **getpwent**. **getpwent** is provided mainly as an aid to porting existing UNIX programs that use it. △

RETURN VALUE

getpwent returns a pointer to a **struct passwd** if successful, and the end of the user database has not been reached. **getpwent** returns **0** if the end of the database has been reached, or if an error condition occurred. See the function description for **getpwuid** in the " **posix** Function Reference" section of the *SAS/C Library Reference, Volume 2* for more information about the definition of **struct passwd**.

Note: The pointer returned by **getpwent** may be a static data area that can be rewritten by the next call to **getpwent**, **getpwuid** or **getpwnam**. △

USAGE NOTES

The **getpwent** function can only be used with MVS 5.2.2 or a later release.

RELATED FUNCTIONS

endpwent, **getpwnam**, **getpwuid**, **setpwent**

getrlimit

Obtain OpenEdition resource limits

SYNOPSIS

```
#include <sys/resource.h>
int getrlimit(int resource,
              struct rlimit *info);
```

DESCRIPTION

The **getrlimit** function determines the resource limits for the calling process. The limits are expressed as a pair of integers, a soft limit and a hard limit. The soft limit controls the amount of the resource the program is actually allowed to consume, while the hard limit specifies an upper bound on the soft limit. The soft limit can be raised up to the hard limit value, but the hard limit can only be raised by a privileged (superuser) caller.

The **info** argument is a pointer to a structure of type **struct rlimit**, into which will be stored the defined limits. The structure contains the following fields:

- **rlim_cur** – the current (i.e., soft) limit
- **rlim_max** – the maximum limit

The **resource** argument defines the resource to which the limit applies. It should be specified as one of the symbolic values defined below.

- **RLIMIT_AS** – The maximum size in bytes of the address space for this process.
- **RLIMIT_CORE** – The maximum size in bytes of an OpenEdition memory dump (core file) for this process.
- **RLIMIT_CPU** – The maximum CPU time in seconds allowed for this address space.
- **RLIMIT_DATA** – The maximum amount of memory available for data allocation using **malloc** or **calloc**. This limit is not enforced under OpenEdition MVS, and an attempt to set the limit lower than **RLIM_INFINITY** will be rejected.
- **RLIMIT_FSIZE** – The maximum file size in bytes allowed for this process. The limit applies only to OpenEdition HFS files, not to standard MVS data sets.
- **RLIMIT_NOFILE** – The maximum number of file descriptors the process may have open.
- **RLIMIT_STACK** – The maximum amount of memory available for stack allocation. This limit is not enforced under OpenEdition MVS, and an attempt to set the limit lower than **RLIM_INFINITY** will be rejected.

RETURN VALUE

getrlimit returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **getrlimit** function can only be used with MVS 5.2.2 or a later release.

Note: Resource limits are inherited from the parent process when **fork** or **exec** is used. △

RELATED FUNCTIONS

setrlimit

getrusage

Obtain OpenEdition usage statistics

SYNOPSIS

```
#include <sys/resource.h>
int getrusage(int type,
              struct rusage *info);
```

DESCRIPTION

The **getrusage** function returns resource usage information for the calling process, or for terminated child processes of the calling process.

The **type** argument is a symbolic constant which specifies the processes for which resource information is wanted. The permissible values are:

- **RUSAGE_SELF** – the current process
- **RUSAGE_CHILDREN** – terminated child processes

Note: If **RUSAGE_CHILDREN** is specified, information is returned only for child processes for which the parent has waited. △

The **info** argument specifies a pointer to an **rusage** structure in which resource usage information is to be stored. The **rusage** structure contains two fields:

1. **ru_utime** – the user time consumed by the process(es)
2. **ru_stime** – the system time consumed by the process(es)

The **ru_utime** and **ru_stime** fields both have type **struct timeval**, which allows a time value to be specified to an accuracy of a microsecond. The structure has two fields:

1. **tv_sec** – the number of seconds used
2. **tv_usec** – the number of microseconds used

RETURN VALUE

getrusage returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **getrusage** function can only be used with MVS 5.2.2 or a later release

getsid

Get session leader id for a process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
pid_t getsid(pid_t pid);
```

DESCRIPTION

The **getsid** function returns the process id for the session leader for a specific process. The argument **pid** should be either the id of a specific process or **0** to request the id of the session leader for the current process.

RETURN VALUE

getsid returns the session leader's process id, or **-1** if it fails.

USAGE NOTES

The **getsid** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE

This example is compiled using **sascc370 -Krent -o**.

```
/*--------------------------------------+
| POSIX/UNIX header files               |
+--------------------------------------*/
#include <sys/types.h>

#include <unistd.h>

/*--------------------------------------+
| ISO/ANSI header files                 |
+--------------------------------------*/
#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

/*--------------------------------------+
| Name:      main                       |
| Returns:   exit(EXIT_SUCCESS)         |
|            or exit(EXIT_FAILURE)      |
+ -------------------------------------*/
int main()
{
/* current process id from getpid()     */
   pid_t pid;
/* process group id from getsid()       */
   pid_t sid;

/*--------------------------------------*/
/* Get the Session Leader id for the    */
/* Current Process                      */
/*--------------------------------------*/
printf("\nGet the Session Leader ID
    of the Current Process\n");

/* version 1 */
/* Set errno to zero for error handling    */
   errno = 0;

/* get the process id for this process     */
```

```
   pid = getpid();

/* get the session leader id for           */
/* this process                            */
   sid = getsid(pid);

/* Test for Error */
   if (sid == -1)
   {
      perror("Call to getsid failed");
      exit(EXIT_FAILURE);
   }
   else
   {
printf
 ("       The process id: %d\n", (int)pid);
printf
 ("The Session Leader id: %d\n", (int)sid);
   }

/* version 2                               */
/* Reset errno to zero for error handling  */
   errno = 0;

 /* 0 implies current processs id          */
   pid = 0;

/* get the session leader id for           */
/* this process                            */
   sid = getsid(pid);

/* Test for Error */
   if (sid == -1)
   {
      perror("Call to getsid failed");
      exit(EXIT_FAILURE);
   }
   else
   {
 printf("       The process id: %d\n",
  (int)pid);
 printf("The Session Leader id: %d\n",
  (int)sid);
   }

   exit(EXIT_SUCCESS);

}  /* end of main() */
```

RELATED FUNCTIONS
setsid

getwd

Get working directory path name

SYNOPSIS

```
#include <unistd.h>
char *getwd(char *buffer);
```

DESCRIPTION
getwd determines and stores the name of the working OpenEdition directory. The **buffer** argument is an array where the path name is to be stored. It should address an area of at least **PATH_MAX+1** bytes.

Note: The **PATH_MAX** symbol is defined in the header file. △

When you call **getwd** in an application which was not compiled with the **posix** option, the returned directory name will begin with an **hfs:** style prefix.

RETURN VALUE
If successful, **getwd** returns a pointer to the **buffer** in which the directory path name was stored. If it fails, it returns **NULL**.

USAGE NOTES
The **getwd** function can only be used with MVS 5.2.2 or a later release.

Note: Use of the more portable and standard **getcwd** function is recommended in place of **getwd**. △

RELATED FUNCTIONS
getcwd

lchown

Change file or link ownership

SYNOPSIS

```
#include <unistd.h>
int lchown(const char *pathname,
           uid_t owner, gid_t group);
```

DESCRIPTION
lchown changes the owner or owning group of an HFS file or symbolic link. It can be used with either regular files or special files such as directories, FIFO files or links. **pathname** is the name of the file or link, **owner** is the new owning user id, and **group** is the new group id. If either **owner** or **group** is specified as -1, the owner or group of the file or link is unchanged.

For programs not compiled with the **posix** option, a style prefix may be required as part of the **pathname** specification. See "File Naming Conventions" in the *SAS/C Library Reference, Volume 1*, for further information.

Note: For information when OpenEdition permits file ownership to be changed, see the **chown** function in the *SAS/C Library Reference, Volume 2*. △

RETURN VALUE
lchown returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES
The **lchown** function can only be used with MVS 5.2.2 or a later release.

RELATED FUNCTIONS
chown, **fchown**

mmap

Map an HFS file into memory

SYNOPSIS

```
#include <mman.h>
void *mmap(void *addr, size_t len,
           int prot, int flags,
           int fd, off_t offset);
```

DESCRIPTION

The **mmap** function is used to request memory mapping of all or part of an HFS file. Memory mapping allocates a block of memory so that fetching from the memory block will obtain the corresponding bytes of the file. Depending on the **mmap** flags, memory mapping is capable of changing the corresponding bytes of the file when data is stored in the memory block.

The **addr** argument may be either **0** or a memory address. If **addr** is **0,** the system will map the file anywhere in memory it chooses. If **addr** is not **0**, the system will attempt to allocate the memory for the file mapping near the address specified.

The **len** argument specifies the number of bytes of the file that are to be mapped. The length must not cause the map to extend beyond the end of the file. If the length does not specify an integral number of pages, the map is extended to the next highest page boundary, and the additional space is set to binary zeroes.

The **prot** argument specifies the protection status of the mapped memory. It should be specified as one of the symbolic constants:

- **PROT_EXEC** – the memory is execute mode only (treated as **PROT_READ** by OpenEdition MVS)
- **PROT_NONE** – no access to the memory is permitted
- **PROT_READ** – the memory can be read, but not written
- **PROT_WRITE** – both read and write access to the memory is permitted

You cannot specify **PROT_WRITE** if the file descriptor for the file does not permit writing.

Note: The protection status of the mapped memory can be changed later by a call to the **mprotect** function. △

The **flags** argument specifies one or more option flags, combined using the or operator (|). Each flag should be specified as one of the following symbolic constants:

- **MAP_SHARED** – the mapped memory is shared with the file, that is, changes to the memory will ultimately be reflected in the file contents (mutually exclusive with **MAP_PRIVATE**)
- **MAP_PRIVATE** – the mapped memory is private, and changes will not be reflected in the file contents (mutually exclusive with **MAP_SHARED**)
- **MAP_FIXED** – the system is required to allocate the memory in the address specified by the **addr** argument

The **fd** argument specifies an open file descriptor for the file to be mapped, which must be a regular HFS file.

The **offset** argument specifies the first byte of the file to be mapped. The offset must be an exact multiple of the system page size (4096 bytes).

Note: See the *IBM OpenEdition Assembler Callable Services* manual for additional information about the behavior of **mmap** and the conditions under which it can be used. △

RETURN VALUE

mmap returns the address of the start of the mapped memory if successful, or **0** if unsuccessful.

USAGE NOTES

The **mmap** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE

This example is compiled using **sascc370 -Krent -o** . The program uses the memory map functions, **mmap()**, **mprotect()**, **msync()**, and **munmap()**, to copy a file similar to the unix **cp** command.

```
/*--------------------------------------+
| POSIX/UNIX header files               |
+--------------------------------------*/
#include <unistd.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/mman.h>

/*--------------------------------------+
| ISO/ANSI header files                 |
+--------------------------------------*/
#include <stdlib.h>

#include <stdio.h>

/*--------------------------------------+
| Types                                 |
+--------------------------------------*/

/*--------------------------------------+
| Constants                             |
+--------------------------------------*/
#define F_MODE
 (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

/*--------------------------------------+
| Name:      main                       |
| Returns:   exit(EXIT_SUCCESS) or      *
|            exit(EXIT_FAILURE)         |
+--------------------------------------*/
int main( int argc, char **argv )
```

```
{
/* file descripter for the src file    */
    int fdSrc;

/* file descripter for the dest file   */
    int fdDest;

/* memory addr of the mapped src file  */
    char *src;

/* memory addr of the mapped dest file */
    char *dest;

/* Buffer for fstat() info             */
    struct stat statBuffer;

/*-------------------------------------*/
/* test args to main                   */
/*-------------------------------------*/
   if ( argc != 3 )
   {
      fprintf(stderr, "Usage: mmcp  \n");
      exit(EXIT_FAILURE);
   }

/*-------------------------------------*/
/* open the source file for read     */
/*-------------------------------------*/
   if ( (fdSrc = open(argv[1], O_RDONLY)) < 0 )
   {
    fprintf(stderr,
     "ERROR: Cannot open source file <%s>\n",
        argv[1]);
       exit(EXIT_FAILURE);
   }

/*-------------------------------------*/
/* open/create new file for output    */
/*-------------------------------------*/
if ( (fdDest =
open(argv[2], O_RDWR | O_CREAT
 | O_TRUNC, F_MODE)) < 0)
   {
 fprintf(stderr,
  "ERROR:
   Cannot create dest file <%s>\n", argv[1]);
    exit(EXIT_FAILURE);
   }

/*-------------------------------------*/
/* retrieve size of source file from    */
/* fstat() function                    */
/*-------------------------------------*/
   if ( fstat(fdSrc, &statBuffer) < 0 )
   {
   fprintf(stderr,
   "ERROR:
   Cannot fstat source file <%s>\n", argv[1]);
   exit(EXIT_FAILURE);
   }

/*-------------------------------------------*/
/* set the size of the dest file.             */
/* If we don't we'll get an                    */
/* SIGSEGV when we try to copy to its mmap.    */
/*-------------------------------------------*/
 if ( ftruncate( fdDest,
  (off_t)statBuffer.st_size ) < 0 )
   {
   fprintf(stderr,
   "ERROR:
   Cannot set size of the dest file <%s>\n",
   argv[2]);
   exit(EXIT_FAILURE);
   }

/*-------------------------------------------*/
/* map the source file to memory               */
/* (for read only)                             */
/*-------------------------------------------*/
if ( (src = mmap(0,
     statBuffer.st_size, PROT_READ, MAP_SHARED,
        fdSrc, 0)) == (char *)-1)
   {
   fprintf(stderr,
   "ERROR: Cannot memory map source file <%s>\n",
   argv[1]);
   exit(EXIT_FAILURE);
   }
   else
   {
   printf
   ("NOTE: source file <%s> mapped
   at address <%p>\n", argv[1], src);
   }

/*-------------------------------------------*/
/* map the dest file to memory                 */
/* (for read only)                             */
/*-------------------------------------------*/
 if ( (dest = mmap
  (0, statBuffer.st_size, PROT_READ,
          MAP_SHARED, fdDest, 0)) == (char *)-1)
   {
   fprintf(stderr,
   "ERROR:
   Cannot memory map dest file <%s>\n", argv[2]);
   exit(EXIT_FAILURE);
   }
   else
   {
   printf("NOTE:
   dest file <%s> mapped at address <%p>\n",
   argv[2], dest);
   }

/*-------------------------------------------*/
/* chg the protection of memory map of         */
/* the dest file to write                      */
/*-------------------------------------------*/
if ( (mprotect(dest,
  statBuffer.st_size, PROT_WRITE)) == -1)
```

```
   {
   fprintf(stderr,
   "ERROR:
   Cannot change memory map protect of
   dest file <%s>\n", argv[2]);
   exit(EXIT_FAILURE);
   }

/*-----------------------------------------*/
/* copy src to dest in memory              */
/*-----------------------------------------*/
   memcpy(dest, src, statBuffer.st_size);

/*-----------------------------------------*/
/* synchronize the memory mapped dest file.  */
/*-----------------------------------------*/
 if ( (msync
  (dest, statBuffer.st_size, MS_SYNC)) == -1)
   {
   fprintf(stderr,
   "ERROR:
   Cannot synchronize memory map of
   dest file <%s>\n", argv[2]);
   exit(EXIT_FAILURE);
   }

/*-----------------------------------------*/
/* cancel mapping of memory to the src file.  */
/*-----------------------------------------*/
   if ( (munmap(src, statBuffer.st_size)) == -1)
   {
   fprintf(stderr,
   "ERROR:
   Cannot terminate memory map of
   src file <%s>\n", argv[2]);
   exit(EXIT_FAILURE);
   }

/*-----------------------------------------*/
/* cancel mapping of memory to the dest file. */
/*-----------------------------------------*/
   if ( (munmap(dest, statBuffer.st_size)) == -1)
   {
   fprintf(stderr,
   "ERROR:
   Cannot terminate memory map of
   dest file <%s>\n", argv[2]);
   exit(EXIT_FAILURE);
   }

   exit(EXIT_SUCCESS);

}  /* end of main() */

/* end of mmcp.c */
```

RELATED FUNCTIONS

mprotect, **msync**, **munmap**

mprotect

Change protection of memory mapped to a file

SYNOPSIS

```
#include <mman.h>
int mprotect(void *addr,
             unsigned int len,
             int prot);
```

DESCRIPTION

The **mprotect** function is used to change the protection status of one or more pages of memory mapped to an HFS file by a previous call to the **mmap** function.

The **addr** argument is the address of the first page of mapped memory for which the protection attributes are to be changed. The address must be on a page boundary, but need not be the first byte of the entire area mapped to a file.

The **len** argument specifies the number of bytes of memory whose protection attributes are to be changed. If the length does not specify an integral number of pages, it is rounded up to do so. The length need not specify the entire area of mapped memory.

The **prot** argument specifies the new protection status of the mapped memory. It should be specified as one of the symbolic constants described in the **mmap** function write-up.

You cannot specify **PROT_WRITE** if the file descriptor for the file does not permit writing.

Note: See the *IBM OpenEdition Assembler Callable Services* manual for additional information about the behavior of **mprotect** and the conditions under which it can be used. △

RETURN VALUE

mprotect returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **mprotect** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE

Refer to “mmap” on page 20 for an example.

RELATED FUNCTIONS

mmap, **msync**, **munmap**

msgctl

Control a message queue

SYNOPSIS

```
#include <sys/msg.h>
int msgctl(int id, int cmd,
           struct msgid_ds *buf);
```

DESCRIPTION

The **msgctl** function is used to perform one of several control operations on an OpenEdition message queue.

Note: See the **msgget** function description in section "msgget" on page 26 for general information about message queues. △

The **id** argument to **msgctl** specifies a message queue id. This argument is an id, such as the id returned by **msgget**, not a message queue key, which might be passed as an argument to **msgget**.

The **cmd** argument should be specified as a symbolic constant specifying the particular operation to be performed by **msgctl**. The constant values are described below.

Several of the **msgctl** operations allow you to obtain or access the message queue id data structure, which is mapped by the **struct msqid_ds** type defined in **sys/msg.h**. This data structure is defined as follows:

```
struct msqid_ds {
   /* permission information */
   struct ipc_perm msg_perm;

   /* messages presently on the queue */
   unsigned msg_qnum;

   /* max bytes of queued info */
   unsigned msg_qbytes;

   /* process id of last sender */
   pid_t msg_lspid;

   /* process id of last receiver */
   pid_t msg_lrpid;

   /* time of last msgsnd call */
   time_t msg_stime;

   /* time of last msg(x)rcv call */
   time_t msg_rtime;

   /* time of last change by msgget/msgctl */
   time_t msg_ctime;
};
```

The **ipc_perm** structure, which contains security information about the owner and the creator of the message queue, is defined as follows:

```
struct ipc_perm {
   /* owner's effective user ID */
   uid_t uid;

   /* owner's effective group ID */
   gid_t gid;

   /* creator's effective user ID */
   uid_t cuid;

   /* creator's effective group ID */
   gid_t cgid;

   /* read/write permission bits */
   mode_t mode;
};
```

For **msgctl** operations which access or modify the message queue data structure, the **buf** argument addresses a **struct msqid_ds**, used as described below. For other operations, the **buf** argument is ignored.

The **cmd** values accepted by **msgctl** and their meanings are as follows:

- **IPC_RMID** – Removes the message queue and its id from the system. The **buf** argument is not used by this operation.
- **IPC_SET** – Can be used to change the ownership of a message queue or the access rules. The contents of **buf->msg_perm.uid**, **buf->msg_perm.gid** and **buf->msg_perm.mode** will be copied to the message queue id data structure.
- **IPC_STAT** – Returns the contents of the message queue id data structure. All elements of the data structure are stored in the object addressed by **buf**.

RETURN VALUE

msgctl returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **msgctl** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE 1

This example is compiled using **sascc370 -Krent -o**. This program uses the functions **msgget()**, **msgsnd()**, **msgrcv()**, and **msgctl()** to establish an IPC Server using Message Queues.

```
/*-------------------------+
| POSIX/UNIX header files  |
+-------------------------*/
#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/ipc.h>

#include <sys/msg.h>

/*-------------------------+
| ISO/ANSI header files    |
+-------------------------*/
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

/*-------------------------+
| Constants                |
+-------------------------*/
/* maximum message size           */
#define MAX_MSGSIZE          256
```

```
/* message key, set by server     */
#define MSG_KEY      (key_t)1097

/* Server's message type          */
#define SERVER_MSG_TYPE (long)10

/* Client's message type          */
#define CLIENT_MSG_TYPE (long)20

/* give everyone read/write        */
/* permission to messages          */
#define MSG_PERM  (S_IRUSR|S_IWUSR
|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

/*---------------------------------+
| Types                            |
+---------------------------------*/
/* Declare the message structure.   */
typedef struct Message
{
    /* positive message type        */
    long    type;
    /* message data                 */
    char    text[MAX_MSGSIZE];
}Message;

/*---------------------------------+
| Name:      main                  |
| Returns:   exit(EXIT_SUCCESS) or |
|            exit(EXIT_FAILURE)   |
+---------------------------------*/
int main()
{
/* message queue id                */
    int msgQID;

/* message flags                    */
    int msgFlags;

/* message type                     */
    long msgType;

/* message key                      */
    key_t msgKey;

/* message to send                  */
    Message sendMsg;

/* message received                 */
    Message recvMsg;

/*----------------------------------*/
/* Create message queue.            */
/* Give everyone read/write         */
/* permissions.                     */
/*----------------------------------*/
   msgKey = MSG_KEY;
   msgFlags = IPC_CREAT | MSG_PERM;

   if ( (msgQID = msgget(msgKey, msgFlags)) < 0 )
   {
       perror("SERVER: msgget");
       exit(EXIT_FAILURE);
   }

/*----------------------------------------*/
/* Receive a message from client.         */
/*----------------------------------------*/
   msgType = CLIENT_MSG_TYPE;
   msgFlags = 0;

   if (msgrcv(msgQID, &recMsg, MAX_MSGSIZE,
       msgType, msgFlags) < 0)
   {
       perror("SERVER: msgrcv");
       msgctl(msgQID, IPC_RMID, NULL);
       exit(EXIT_FAILURE);
   }

/*----------------------------------------*/
/* Print message received from client.    */
/*----------------------------------------*/
   printf("%s\n", recvMsg.text);

/*----------------------------------------*/
/* Send ACK Message to client.            */
/*----------------------------------------*/
sendMsg.type = SERVER_MSG_TYPE;
sprintf(sendMsg.text,
        "From SERVER: Message received!");
msgFlags = 0;

   if (msgsnd(msgQID, &sendMsg,
   strlen(sendMsg.text)+1, msgFlags) < 0)
   {
       perror("SERVER: msgsnd");
       msgctl(msgQID, IPC_RMID, NULL);
       exit(EXIT_FAILURE);
   }

/*----------------------------------------*/
/* Go to sleep to allow time for the client  */
/* to recieve the message                    */
/* before removing the message queue.        */
/*----------------------------------------*/
   sleep(60);

/*----------------------------------------*/
/* Call msgctl to remove message queue.   */
/*----------------------------------------*/
   if (msgctl(msgQID, IPC_RMID, NULL) < 0)
   {
       perror("SERVER: msgctl");
       exit(EXIT_FAILURE);
   }

   exit(EXIT_SUCCESS);

}
/* end of main() */
```

EXAMPLE 2

This example is compiled using **sascc370 -Krent -o.**

This program usese the functions **msgctl()**, **msgget()**, **msgsnd()**, and **msgxrcv** to establish IPC Client using XMessage Queues. Also, it uses Open Edition's extended Message structure.

Note: You cannot use the Extended Message structure to send a message, i.e., call **msgsnd()** with this structure. Attempting to do so will cause the program to "hang". △

```
/*-------------------------------------------+
| POSIX/UNIX header files                    |
+-------------------------------------------*/
#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/ipc.h>

#include <sys/msg.h>

/*-------------------------------------------+
| ISO/ANSI header files                      |
+-------------------------------------------*/
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <time.h>

#include <errno.h>

/*-------------------------------------------+
| Constants                                  |
+-------------------------------------------*/
 /* maximum message size                     */
#define MAX_MSGSIZE          256

/* message key, set by server                */
#define MSG_KEY      (key_t)1097

/* Server's message type                     */
#define SERVER_MSG_TYPE (long)10

/* Client's message type                     */
#define CLIENT_MSG_TYPE (long)20


/* give everyone read/write                  */
/* permission to messages                    */
#define MSG_PERM
(S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

/*-------------------------------------------+
| Types                                      |
+-------------------------------------------*/
/* Declare the message structure. */
typedef struct Message
{
/* positive message type          */
    long    type;
/* message data                   */
    char    text[MAX_MSGSIZE];
}Message;

/* Declare the message structure. */
typedef struct XMessage
{
/* time msg sent                  */
    time_t  time;

/* effective uid of sender        */
    uid_t   uid;

/* effective gid of sender        */
    gid_t   gid;

/* process id of sender           */
    pid_t   pid;

/* positive message type          */
    long    type;

/* message data                   */
    char    text[MAX_MSGSIZE];
}XMessage;

/*-------------------------------------------+
| Name:      main                            |
| Returns:   exit(EXIT_SUCCESS) or           |
|            exit(EXIT_FAILURE)              |
+-------------------------------------------*/
int main()
{
/* message queue id                          */
    int msgQID;

/* message flags                             */
    int msgFlags;

/* command to message queue, used w/ msgctl *
    int msgQcmd;          /

/* message type                              */
    long msgType;

/* message key                               */
    key_t msgKey;

/* message to send                           */
    Message sendMsg;

/* message received - eXtended               */
    XMessage recvMsg;

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

```
/* Create message queue.                    */
/* Give everyone read/write permissions.    */
/*------------------------------------------*/
   msgKey = MSG_KEY;
   msgFlags = IPC_CREAT | MSG_PERM;

 if ( (msgQID = msgget(msgKey, msgFlags)) < 0 )
   {
       perror("SERVER: msgget");
       exit(EXIT_FAILURE);
   }

/*------------------------------------------*/
/* Receive a message from client.           */
/*------------------------------------------*/
   msgType = CLIENT_MSG_TYPE;
   msgFlags = 0;

   if (msgxrcv(msgQID, &recMsg, MAX_MSGSIZE,
         msgType, msgFlags) < 0)
   {
       perror("SERVER: msgrcv");
       msgctl(msgQID, IPC_RMID, NULL);
       exit(EXIT_FAILURE);
   }

/*------------------------------------------*/
/* Print message received from client.      */
/*------------------------------------------*/
   printf("Message: %s\n", recvMsg.text);
   printf("   Time message was sent: %s\n",
     ctime(&recvMsg.time));
   printf("   The user id of sender: %d\n",
     (int)recvMsg.uid);
   printf("  The group id of sender: %d\n",
     (int)recvMsg.gid);
   printf("The process id of sender: %d\n",
     (int)recvMsg.pid);

   /*---------------------------------------*/
   /* Send ACK XMessage to client.          */
   /*---------------------------------------*/
   sendMsg.type = SERVER_MSG_TYPE;
   sprintf(
    sendMsg.text,
     "From SERVER: XMessage received!");
   msgFlags = 0;

   if (msgsnd(msgQID, &sendMsg,
      strlen(sendMsg.text)+1, msgFlags) < 0)
   {
       perror("SERVER: msgsnd");
       msgctl(msgQID, IPC_RMID, NULL);
       exit(EXIT_FAILURE);
   }

/*------------------------------------------*/
/* Go to sleep to allow time for the client to  */
/* recieve the message                      */
/* before removing the message queue.       */
   /*---------------------------------------*/
      sleep(60);

   /*---------------------------------------*/
   /* Call msgctl to remove message queue.  */
   /*---------------------------------------*/
      if (msgctl(msgQID, IPC_RMID, NULL) < 0)
      {
      perror("SERVER: msgctl");
      exit(EXIT_FAILURE);
      }

      exit(EXIT_SUCCESS);

   }  /* end of main() */
```

RELATED FUNCTIONS

msgget, **msgrcv**, **msgsnd**, **msgxrcv**

msgget

Create or find a message queue

SYNOPSIS

```
#include <sys/msg.h>

int msgget(key_t key, int flags);
```

DESCRIPTION

The **msgget** function is used either to create a new message queue or to locate an existing queue based on a key. Message queues are implemented by OpenEdition, and allow messages of multiple types to be queued. OpenEdition message queues support multiple senders and multiple receivers and do not require the senders and receivers to be running simultaneously. Message queues, once created using **msgget**, remain in existence until they are explicitly destroyed with a call to **msgctl**.

The **key** argument is an integral value which identifies the message queue desired. A **key** value of **IPC_PRIVATE** requests a new message queue without an associated **key**, and which can be accessed only by the queue id returned by **msgget**.

The **flags** argument specifies zero or more option flags specifying whether or not the queue already exists, and how access to the queue should be regulated. The argument should be specified as **0** for no flags, or as one or more of the following symbolic constants, combined using the or operator (|):

- **IPC_CREAT** – Specifies that if a queue with the requested **key** does not exist, it should be created. This flag is ignored if **IPC_PRIVATE** is specified.
- **IPC_EXCL** – Specifies that a queue with the requested **key** must not already exist. This flag is ignored if **IPC_PRIVATE** is specified, or if **IPC_CREAT** is not specified.

Additionally, any of the permission bits **S_IRUSR**, **S_IWUSR**, **S_IRGRP**, **S_IWGRP**, **S_IROTH** and **S_IWOTH** may be specified as part of the **flags** argument, to regulate what users are permitted to access or modify

the message queue. See the **umask** function description in the *SAS/C Library Reference, Volume 2*, for more information about the meaning of these flags.

RETURN VALUE

msgget returns the identifier of the message queue if successful, or **-1** if unsuccessful.

USAGE NOTES

The **msgget** function can only be used with MVS 5.2.2 or a later release.

Note: A site can impose limits on the number and size of queued messages. △

EXAMPLE 1

This example is compiled using **sascc370 -Krent -o**. This program uses the functions **msgget()**, **msgsnd()**, and **msgrcv()** to establish IPC Client using Message Queues.

```
/*-----------------------------------------+
| POSIX/UNIX header files                  |
+-----------------------------------------*/
#include <fcntl.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

/*-----------------------------------------+
| ISO/ANSI header files                    |
+-----------------------------------------*/
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

/*-----------------------------------------+
| Constants                                |
+-----------------------------------------*/
/* maximum message size           */
#define MAX_MSGSIZE          256

/* message key, set by server     */
#define MSG_KEY      (key_t)1097

/* Server's message type          */
#define SERVER_MSG_TYPE (long)10

/* Client's message type          */
#define CLIENT_MSG_TYPE (long)20

/* give everyone read/write       */
/* permission to messages         */
#define MSG_PERM  (S_IRUSR|S_IWUSR|S_IRGRP
  |S_IWGRP|S_IROTH|S_IWOTH)

/*---------------------------------+
| Types                            |
+---------------------------------*/
/* Declare the message structure. */
typedef struct Message
{
/* positive message type          */
    long    type;
/* message data                   */
    char    text[MAX_MSGSIZE];
}Message;

/*---------------------------------+
| Name:      main                  |
| Returns:   exit(EXIT_SUCCESS) or |
|            exit(EXIT_FAILURE)    |
+---------------------------------*/
int main()
{
/* message queue id                          */
    int msgQID;

/* message flags                             */
    int msgFlags;

/* message type                              */
    long msgType;

/* message key                               */
    key_t msgKey;

/* message to send                           */
    Message sendMsg;

/* message received                          */
    Message recvMsg;

/*-----------------------------------------*/
/* Get the message queue id for MSG_KEY,    */
/* which was set by the                     */
/* message server.                          */
/*-----------------------------------------*/
   msgKey = MSG_KEY;
   msgFlags = MSG_PERM;

   if ( (msgQID = msgget(msgKey, msgFlags)) < 0 )
   {
       perror("CLIENT: msgget");
       exit(EXIT_FAILURE);
   }

/*-----------------------------------------*/
/* Send the message.                        */
/*-----------------------------------------*/
   sendMsg.type = CLIENT_MSG_TYPE;
   sprintf
    (sendMsg.text, "From CLIENT: Are you there?");
   msgFlags = 0;

   if (msgsnd(msgQID, &sendMsg,
     strlen(sendMsg.text)+1, msgFlags) < 0)
```

```
   {
   perror("CLIENT: msgsnd");
   exit(EXIT_FAILURE);
   }

/*-----------------------------------------*/
/* Receive a message from server.          */
/*-----------------------------------------*/
   msgType = SERVER_MSG_TYPE;
   msgFlags = 0;

   if (msgrcv(msgQID, &recMsg, MAX_MSGSIZE,
     msgType, msgFlags) < 0)
   {
   perror("CLIENT: msgrcv");
   exit(EXIT_FAILURE);
   }

/*-----------------------------------------*/
/* Print message received from server.     */
/*-----------------------------------------*/
   printf("%s\n", recvMsg.text);

   exit(EXIT_SUCCESS);

}  /* end of main() */
```

EXAMPLE 2

Refer to "msgctl" on page 22 for an example program that establishes an IPC Server using Message Queues.

EXAMPLE 3

This example is compiled using **`sascc370 -Krent -o`**. This program uses the **`msgget()`** function to create IPC message queue and then uses the **`msgctl()`** function to retrieve statistics on the newly created message queue.

```
/*-----------------------------------------+
| POSIX/UNIX header files                  |
+-----------------------------------------*/
#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/ipc.h>

#include <sys/msg.h>

/*-----------------------------------------+
| ISO/ANSI header files                    |
+-----------------------------------------*/
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <time.h>

#include <errno.h>

/*-----------------------------------------+
| Types                                    |
+-----------------------------------------*/
/* message queue id structure     */
typedef struct msqid_ds MsgQueueID;

/*-----------------------------------------+
| Constants                                |
+-----------------------------------------*/
/* message key                    */
#define MSG_KEY      (key_t)1097

/* give everyone read/write permission to     */
/*messages                                    */
#define MSG_PERM (S_IRUSR|S_IWUSR|S_IRGRP
  |S_IWGRP|S_IROTH|S_IWOTH)

/*-----------------------------------------+
| Name:      main                          |
| Returns:   exit(EXIT_SUCCESS) or         |
|            exit(EXIT_FAILURE)            |
+-----------------------------------------*/
int main()
{
/* message queue id                  */
    int msgQID;

/* message flags                     */
    int msgFlags;

/* command to message queue.         */
    int msgQcmd;

/* message key                       */
    key_t msgKey;

/* message queue id data structure    */
    MsgQueueID * msgQueue;

/*-------------------------------------------*/
/* Create message queue.                     */
/* Give everyone read/write permissions.     */
/*-------------------------------------------*/
   msgKey = MSG_KEY;
   msgFlags = IPC_CREAT | MSG_PERM;

   if ( (msgQID = msgget(msgKey, msgFlags)) < 0 )
   {
       perror("MSGSTAT: msgget");
       exit(EXIT_FAILURE);
   }

/*-------------------------------------------*/
/* Allocate memory to store information from     */
/* message queue                             */
/*-------------------------------------------*/
   msgQueue = malloc(sizeof(MsgQueueID));

   if (msgQueue == NULL)
   {
```

```
   fprintf(stderr,
   "ERROR:
   Cannot allocate memory for message queue
   stats\n");
   exit(EXIT_FAILURE);
   }

/*-----------------------------------------------*/
/* Call msgctl to retrieve information from      */
/* message queue                                 */
   /*-----------------------------------------*/
   msgQcmd = IPC_STAT;

   if (msgctl(msgQID, msgQcmd, msgQueue) < 0)
   {
   perror
   ("MSGSTAT: msgctl failed to retrieve
   message queue stats");
   free(msgQueue);
   exit(EXIT_FAILURE);
   }

/*-----------------------------------------------*/
/* Print infomation retrieved from message queue */
/*-----------------------------------------------*/
   printf("Message Queue ID statistics:\n\n");
   printf("\tQueue owner's effective user ID:
     %d\n",(int)msgQueue->msg_perm.uid);
   printf("\tQueue owner's effective group ID:
     %d\n",(int)msgQueue->msg_perm.gid);
   printf("\tQueue creator's effective user ID:
     %d\n",(int)msgQueue->msg_perm.cuid);
   printf("\tQueue creator's effective group ID:
     %d\n",(int)msgQueue->msg_perm.cgid);
   printf("\tQueue permission mode bits:
     %#.3o\n\n",(int)msgQueue->msg_perm.mode);
   printf("\tNumber of messages currently on Queue:
     %d\n",msgQueue->msg_qnum);
   printf("\tMaximum bytes of queued info:
     %d\n", msgQueue->msg_qbytes);
   printf("\tProcess id of last sender:
     %d\n", (int)msgQueue->msg_lspid);
   printf("\tProcess id of last receiver:
     %d\n", (int)msgQueue->msg_lrpid);
   printf("\tTime of last msgsnd call:
     %s",ctime(&msgQueue->msg_stime));
   printf("\tTime of last msg(x)rcv call:
     %s",ctime(&msgQueue->msg_rtime));
   printf("\tTime of last chg by msgget/msgctl:
     %s",ctime(&msgQueue->msg_ctime));

/*-----------------------------------------------*/
/* Free memory used to store information from    */
/* message queue                                 */
/*-----------------------------------------------*/
   free(msgQueue);

/*-----------------------------------------------*/
/* Call msgctl to remove message queue.          */
/*-----------------------------------------------*/
   msgQcmd = IPC_RMID;
   if (msgctl(msgQID, msgQcmd, NULL) < 0)
   {
   perror
   ("MSGSTAT: msgctl failed to remove message queue");
   exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);

}  /* end of main() */
```

EXAMPLE 4

This example is compiled using **sascc370 -Krent -o**.

This program uses the functions **msgget()**, **msgsnd()**, and **msgxrcv()** to establish IPC Client using XMessage Queues. Also, it uses Open Edition's extended Message structure.

Note: You cannot use the Extended Message structure to send a message, i.e., call **msgsnd()** with this structure. Attempting to do so will cause the program to "hang". △

```
   /*---------------------------------+
   | POSIX/UNIX header files          |
   +---------------------------------*/
   #include <fcntl.h>

   #include <sys/types.h>

   #include <sys/ipc.h>

   #include <sys/msg.h>

   /*---------------------------------+
   | ISO/ANSI header files            |
   +---------------------------------*/
   #include <stdlib.h>

   #include <stdio.h>

   #include <string.h>

   #include <time.h>

   #include <errno.h>

   /*---------------------------------+
   | Constants                        |
   +---------------------------------*/
   /* maximum message size           */
   #define MAX_MSGSIZE          256

   /* message key, set by server     */
   #define MSG_KEY      (key_t)1097

   /* Server's message type          */
   #define SERVER_MSG_TYPE (long)10

   /* Client's message type          */
   #define CLIENT_MSG_TYPE (long)20
```

```
/* give everyone read/write       */
/* permission to messages         */
#define MSG_PERM
 (S_IRUSR|S_IWUSR|S_IRGRP
  |S_IWGRP|S_IROTH|S_IWOTH)

/*-----------------------------------+
| Types                              |
+-----------------------------------*/
/* Declare the message structure. */
typedef struct Message
{
/* positive message type          */
    long    type;
/* message data                   */
    char    text[MAX_MSGSIZE];
}Message;

/* Declare the OE Extended message structure. */
typedef struct XMessage
{
/* time msg sent                  */
    time_t  time;

/* effective uid of sender        */
    uid_t   uid;

/* effective gid of sender        */
    gid_t   gid;

/* process id of sender           */
    pid_t   pid;

/* positive message type          */
    long    type;

/* message data                   */
    char    text[MAX_MSGSIZE];
}XMessage;

/*-----------------------------------+
| Name:      main                    |
| Returns:   exit(EXIT_SUCCESS) or   |
|            exit(EXIT_FAILURE)     |
+-----------------------------------*/
int main()
{
/* message queue id                        */
    int msgQID;

/* message flags                           */
    int msgFlags;

/* message type                            */
    long msgType;

/* message key                             */
    key_t msgKey;

/* message to send                         */
    Message sendMsg;

/* message received - eXtended             */
    XMessage recvMsg;

/*-----------------------------------------*/
/* Get the message queue id for MSG_KEY,   */
/* which was set by the                    */
/* message server.                         */
/*-----------------------------------------*/
   msgKey = MSG_KEY;
   msgFlags = MSG_PERM;

   if ( (msgQID = msgget(msgKey, msgFlags)) < 0 )
   {
       perror("CLIENT: msgget");
       exit(EXIT_FAILURE);
   }

/*-----------------------------------------*/
/* Send the message.                       */
/*-----------------------------------------*/
sendMsg.type = CLIENT_MSG_TYPE;
sprintf
  (sendMsg.text, "From CLIENT: Are you there?");
  msgFlags = 0;

if (msgsnd(msgQID, &sendMsg,
   strlen(sendMsg.text)+1, msgFlags) < 0)
   {
       perror("CLIENT: msgsnd");
       exit(EXIT_FAILURE);
   }

/*-----------------------------------------*/
/* Receive a message from server.          */
/*-----------------------------------------*/
   msgType = SERVER_MSG_TYPE;
   msgFlags = 0;

  if (msgxrcv(msgQID,
   &recMsg, MAX_MSGSIZE, msgType, msgFlags) < 0)
   {
       perror("CLIENT: msgrcv");
       exit(EXIT_FAILURE);
   }

/*-----------------------------------------*/
/* Print message received from server.     */
/*-----------------------------------------*/
   printf("Message: %s\n", recvMsg.text);
   printf("   Time message was sent: %s\n",
     ctime(&recMsg.time));
   printf("   The user id of sender: %d\n",
    (int)recvMsg.uid);
   printf("  The group id of sender: %d\n",
    (int)recvMsg.gid);
   printf("The process id of sender: %d\n",
    (int)recvMsg.pid);

   exit(EXIT_SUCCESS);
```

```
}  /* end of main() */
```

EXAMPLE 5

Refer to "msgctl" on page 22 for an example that establishes an IPC Server using Message Queues. Also, it uses Open Edition's extended Message structure.

RELATED FUNCTIONS

msgctl, **msgrcv**, **msgsnd**, **msgxrcv**

msgrcv

Receive a message from a message queue

SYNOPSIS

```
#include <sys/msg.h>
int msgrcv(int id, void *msg, size_t size,
           long msgtype,int flags);
```

DESCRIPTION

The **msgrcv** function is used to receive a message from an OpenEdition message queue. The **msgtype** argument allows the caller to select the type of message to be received.

Note: See the **msgget** function description in section "msgget" on page 26 for general information about OpenEdition message queues. △

The **id** argument to **msgrcv** specifies the id of the message queue from which messages will be received. This argument is an id, such as the id returned by **msgget**, not a message queue key, which might be passed as an argument to **msgget**.

The **msg** argument should be a pointer to a message buffer into which the received message should be stored. The message buffer should have the following layout:

```
struct {
   long mtype;     /* message type */
   char mtext[n];  /* message text */
};
```

where **n** is an integer constant large enough for the message text.

Note: You must declare an appropriate type for the messages you receive yourself, as no such type is defined by **sys/msg.h**. Also the meaning of **mtype** is not fixed by the message queue protocol, but can be used by the application to establish message types or priorities in any natural way. △

The **size** argument to **msgrcv** should be the maximum number of bytes of message text to be received. **size** should not include the size of the **mtype** field. If the size of the message text to be received is larger than the **size** argument, the result depends upon the options specified by the **flags** argument.

The **msgtype** argument specifies which message, if any, is received. The argument is interpreted as follows:

- If **msgtype** is **0**, the first message on the queue is received.
- If **msgtype** is greater than **0**, the first message whose **mtype** field is equal to **msgtype** is received.
- If **msgtype** is less than zero, a message whose **mtype** field is less than or equal to **-msgtype** is received. If there is more than one such message, the first one of minimal **mtype** value is received.

The **flags** argument specifies zero or more option flags specifying processing options. The argument should be specified as **0** for no flags, or as one or more of the following symbolic constants, combined using the or operator (|):

- **IPC_NOWAIT** – Specifies that the caller does not wish to wait if there is no appropriate message on the queue. In this case, **msgrcv** fails, and **errno** is set to **ENOMSG**. If **IPC_NOWAIT** is not set and there is no appropriate message, the caller will wait until an appropriate message is sent, the message queue is destroyed, or a signal is received.
- **MSG_NOERROR** – Specifies that receiving a message whose text is larger than the **size** argument is not to be considered an error. In this case, the message text will be truncated and no indication of the truncation is returned. If **MSG_NOERROR** is not specified and the message to be received is larger than the **size** argument, the **msgrcv** call fails, and **errno** is set to **E2BIG**.

RETURN VALUE

If successful, **msgrcv** returns the number of bytes of message text stored (not including the **mtype** field). If it fails, **msgrcv** returns **-1**.

USAGE NOTES

The **msgrcv** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE 1

Refer to "msgget" on page 26 for an example.

EXAMPLE 2

Refer to "msgctl" on page 22 for an example that establishes and IPC Server using Message Queues.

RELATED FUNCTIONS

msgctl, **msgget**, **msgsnd**, **msgxrcv**

msgsnd

Send a message to a message queue

SYNOPSIS

```
#include <sys/msg.h>
int msgsnd(int id, const void *msg,
           size_t size, int flags);
```

DESCRIPTION

The **msgsnd** function is used to send a message to an OpenEdition message queue, for later receipt by another process calling **msgrcv** or **msgxrcv**.

Note: See the **msgget** function description in section "msgget" on page 26 for general information about message queues. △

The **id** argument to **msgsnd** specifies the id of the message queue to which messages will be sent. This argument is an id, such as the id returned by **msgget**, not a message queue key, which might be passed as an argument to **msgget**.

The **msg** argument should be a pointer to a message buffer that contains the message to be sent. The message buffer should have the following layout:

```
struct {
   long mtype;     /* message type */
   char mtext[n];  /* message text */
};
```

where **n** is an integer constant large enough for the message text.

Note: You must declare an appropriate type for the messages you send yourself, as no such type is defined by **sys/msg.h**. Also the meaning of **mtype** is not fixed by the message queue protocol (other than that its value must be greater than zero), but it can be used by the application to establish message types or priorities in any natural way. △

The **size** argument to **msgsnd** should be the number of bytes of message text to be sent. **size** should not include the size of the **mtype** field. It is possible to send a message which contains only a type, but no text, in which case a size of **0** should be passed.

The **flags** argument specifies either **0** or the **IPC_NOWAIT** option. If **IPC_NOWAIT** is set, and the system limits on the amount of queued up data have been reached, the call to **msgsnd** will immediately fail, with **errno** set to **EAGAIN**. If **IPC_NOWAIT** is not set, and the limits have been reached, the calling process will wait until enough messages are removed from the queue to allow this message to be sent.

RETURN VALUE

msgsnd returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **msgsnd** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE 1

Refer to "msgget" on page 26 for an example.

EXAMPLE 2

Refer to "msgctl" on page 22 for an example that establishes an IPC Server using Message Queues.

EXAMPLE 3

Refer to "msgget" on page 26 for an example that establishes IPC Client using XMessage Queues.

EXAMPLE 4

Refer to "msgctl" on page 22 for an example that establishes an IPC Server using Message Queues. Also, it uses Open Edition's extended Message structure.

RELATED FUNCTIONS

msgctl, **msgget**, **msgrcv**, **msgxrcv**

msgxrcv

Receive a message from a message queue with sender information

SYNOPSIS

```
#include <sys/msg.h>
int msgxrcv(int id, void *msg,
            size_t size, long msgtype,
            int flags);
```

DESCRIPTION

The **msgxrcv** function is used to receive a message from an OpenEdition message queue. The **msgtype** argument allows the caller to select the type of message to be received. Unlike **msgrcv**, **msgxrcv** stores information about the time the message was sent and the process which sent the message.

Note: See the **msgget** function description in section "msgget" on page 26 for general information about OpenEdition message queues. △

The **id** argument to **msgxrcv** specifies the id of the message queue from which messages will be received. This argument is an id, such as the id returned by **msgget**, not a message queue key, which might be passed as an argument to **msgget**.

The **msg** argument should be a pointer to a message buffer into which the received message should be stored. The message buffer should have the following layout:

```
struct {
   /* the time the message was sent */
   time_t mtime;

   /* the effective uid of the sender */
   uid_t muid;

   /* the effective gid of the sender */
   gid_t mgid;

   /* the process id of the sender */
   pid_t mpid;

   /* message type */
   long mtype;

   /* message text */
   char mtext[n];
};
```

where *n* is an integer constant large enough for the message text.

Note: You must declare an appropriate type for the messages you receive yourself, as no such type is defined by **sys/msg.h**. Also, the meaning of **mtype** is not fixed by the message queue protocol, but can be used by the application to establish message types or priorities in any natural way. △

The **size** argument to **msgxrcv** should be the maximum number of bytes of message text to be received. **size** should not include the size of any of the fields preceding **mtext** in the structure. If the size of the message text to be received is larger than the **size** argument, the result depends upon the options specified by the **flags** argument.

The **msgtype** argument specifies which message, if any, is received. The argument is interpreted as follows:

- If **msgtype** is **0**, the first message on the queue is received.
- If **msgtype** is greater than **0**, the first message whose **mtype** field is equal to **msgtype** is received.
- If **msgtype** is less than zero, a message whose **mtype** field is less than or equal to **-msgtype** is received. If there is more than one such message, the first one of minimal **mtype** is received.

The **flags** argument specifies zero or more option flags specifying processing options. The argument should be specified as **0** for no flags, or as one or more of the following symbolic constants, combined using the or operator (|):

- **IPC_NOWAIT** – Specifies that the caller does not wish to wait if there is no appropriate message on the queue. In this case, **msgxrcv** fails, and **errno** is set to **ENOMSG**. If **IPC_NOWAIT** is not set and there is no appropriate message, the caller will wait until an appropriate message is sent, the message queue is destroyed, or a signal is received.
- **MSG_NOERROR** – Specifies that receiving a message whose text is larger than the **size** argument is not to be considered an error. In this case, the message text will be truncated and no indication of the truncation is returned. If **MSG_NOERROR** is not specified and the message to be received is larger than the **size** argument, the **msgxrcv** call fails, and **errno** is set to **E2BIG**.

RETURN VALUE

If successful, **msgxrcv** returns the number of bytes of message text stored (not including the header fields). If it fails, **msgxrcv** returns **-1**.

USAGE NOTES

The **msgxrcv** function can only be used with MVS 5.2.2 or a later release.

PORTABILITY

msgxrcv is an OpenEdition extension to UNIX message queue processing, and is not portable.

EXAMPLE 1

Refer to "msgget" on page 26 for an example that establishes an IPC Client using XMessage Queues.

EXAMPLE 2

Refer to "msgctl" on page 22 for an example that establishes an IPC Server using Message Queues. Also, it uses Open Edition's extended Message structure.

RELATED FUNCTIONS

msgctl, **msgget**, **msgrcv**, **msgsnd**

msync

Synchronize a memory mapped HFS file

SYNOPSIS

```
#include <mman.h>
int msync(void *addr, unsigned int len,
          int flags);
```

DESCRIPTION

The **msync** function is used to synchronize an HFS file with one or more pages of memory mapped to the file. The call may either cause the file to be updated with data in memory or cause the data in memory to be updated from the file.

The **addr** argument is the address of the first page of mapped memory to be synchronized. The address must be on a page boundary, but need not be the first byte of the entire area mapped to a file.

The **len** argument specifies the number of bytes of memory to be synchronized. If the length does not specify an integral number of pages, it is rounded up to do so. The length need not specify the entire area of mapped memory.

The **flags** argument specifies options describing how synchronization should be performed. One or more of the following symbolic constants should be specified, combined using the or operator (|).

- **MS_ASYNC** – All modified data in the memory specified is written to the file. The writes are performed asynchronously, which means that they need not be performed in any particular order. Control is returned as soon as all writes have been scheduled.
- **MS_SYNC** – All modified data in the memory specified is written to the file. The writes are performed synchronously, that is, control is not returned from **msync** until all data has been written.
- **MS_INVALIDATE** – The contents of the memory pages specified are discarded. References to data in the affected pages will cause data to be read from the appropriate portion of the mapped file.

One of the three flags must be specified, and only one of **MS_ASYNC** and **MS_SYNC** may be specified. If **MS_INVALIDATE** is specified together with another flag, the memory contents specified are discarded only after all write operations have completed.

See the *IBM OpenEdition Assembler Callable Services* manual for additional information about the behavior of **msync** and the conditions under which it can be used.

RETURN VALUE

msync returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **msync** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE

Refer to "mmap" on page 20 for an example.

RELATED FUNCTIONS

mmap, **mprotect**, **munmap**

munmap

Cancel mapping of a memory area to a file

SYNOPSIS

```
#include <mman.h>
int munmap(void *addr, unsigned int len);
```

DESCRIPTION

The **munmap** function is used to terminate mapping of part or all of a memory area previously mapped to a file by the **mmap** function.

The **addr** argument is the address of the first page of mapped memory to be unmapped. The address must be on a page boundary, but need not be the first byte of the entire area mapped to a file.

The **len** argument specifies the number of bytes of memory to be unmapped. If the length does not specify an integral number of pages, it is rounded up to do so. The length need not specify the entire area of mapped memory.

If an area of memory is mapped and then partially unmapped, any reference to an unmapped portion will cause a segmentation violation.

If the memory area was created by a call to **mmap** specifying the **MAP_SHARED** symbolic flag, all changed areas of memory are written back to the file before **munmap** completes. If the **mmap** call specified **MAP_PRIVATE**, all changes are discarded.

See the *IBM OpenEdition Assembler Callable Services* manual for additional information about the behavior of **munmap** and the conditions under which it can be used.

RETURN VALUE

munmap returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **munmap** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE

Refer to "mmap" on page 20 for an example.

RELATED FUNCTIONS

mmap, **mprotect**, **msync**

oetaskctl

Control subtasks with OpenEdition

SYNOPSIS

```
#include <lclib.h>
int oetaskctl(int setting);
```

DESCRIPTION

The **oetaskctl** function is used to control the interpretation of MVS subtasks of the current program, such as subtasks created by the system or **oslink** functions, or by the assembler **ATTACH** macro. By default, subtasks of a program which has used OpenEdition facilities are treated as *threads*. This means that OpenEdition resources, such as file descriptors, current directory, and signal handlers, are shared between the subtasks. Because the SAS/C library assumes that these resources are not shared, this mode of operation can lead to errors.

Alternately, **oetaskctl** can be used to indicate that subtasks of the calling program are to be treated as a separate process from the calling program. This implies each subtask will have its own set of file descriptors, its own current directory, and its own signal handling. This is generally recommended when both tasks are SAS/C programs.

The **setting** argument to **oetaskctl** specifies whether a new subtask should be treated as a thread or a process. An argument of **0** specifies a thread, and **1** specifies a process.

Note: The *thread or process* decision is made when the subtask calls the first OpenEdition function, not when the subtask is **ATTACH**ed. △

RETURN VALUE

oetaskctl returns the previous **oetaskctl** setting if successful. It returns **-1** if it was unable to complete successfully.

USAGE NOTES

oetaskctl is useful only if both the calling task and the subtask use OpenEdition facilities.

SEE ALSO

ATTACH, **system**

readextlink

Read an external link

SYNOPSIS

```
#include <unistd.h>
int readextlink(const char *name,
                char *buf,
                size_t size);
```

DESCRIPTION

readextlink reads the contents of an external link. (See the **extlink** function description in section "extlink" on page 14 for further information on external links.) The **name** argument specifies the name of the external link. The **buf** argument specifies the address of a buffer into which the contents of the link should be read, and **size** specifies the size of the buffer in bytes. If **size** is **0**, no data is stored, and the length required is returned as the function value.

When you call **readextlink** in an application which was not compiled with the

```
posix
```

option, the link name is interpreted according to the normal rules for file name interpretation. For this reason, when not compiled with

```
posix
```

the file name should include a style prefix unless the default style is **hfs**.

Note: The name stored in **buf** will not contain any prefix. △

RETURN VALUE

readextlink returns the number of bytes stored in the buffer, or the number of bytes required if **size** was zero, or **-1** if unsuccessful.

USAGE NOTES

The **getrusage** function can only be used with MVS 5.2.2 or a later release.

RELATED FUNCTIONS

extlink, **lstat**

realpath

Return absolute pathname

SYNOPSIS

```
#include <lclib.h>
char *realpath(const char *pathname,
               char absname[PATH_MAX+1]);
```

If the feature test macro **_SASC_POSIX_SOURCE** is defined and **_POSIX_SOURCE** is not defined, the prototype is also visible in **<stdlib.h>**.

DESCRIPTION

The **realpath** function can be used to return a standard form of an OpenEdition path name. The path name returned will be an absolute path name, which does not involve the "." or ".." notations or symbolic links.

The **pathname** argument to **realpath** is the OpenEdition path name to be resolved. If the program was not compiled with the

```
posix
```

compiler option, **pathname** should begin with an **hfs:** style prefix.

The **absname** argument should be an array of **PATH_MAX**+1 characters, in which **realpath** will store the canonical form of the path name. **PATH_MAX** is defined in the header file **<limits.h>**. If the program was not compiled with the

```
posix
```

compiler option, the value stored will begin with an **hfs:** style prefix.

RETURN VALUE

realpath returns the address of the standardized path name if successful, and otherwise returns **0**.

USAGE NOTES

The **realpath** function can only be used with MVS 5.2.2 or a later release.

semctl

Control a semaphore set

SYNOPSIS

```
#include <sys/sem.h>
int semctl(int id,
           int num,
           int cmd, argument);
```

DESCRIPTION

The **semctl** function is used to perform one of several control operations on an OpenEdition semaphore set.

Note: See the **semget** function description in section "semget" on page 36 for general information about semaphore sets. △

The **id** argument to **semctl** specifies a semaphore set id. This argument is an id, such as the id returned by **semget**, not a semaphore set key, which might be passed as an argument to **semget**.

The **num** argument to **semctl** specifies the index of the specific semaphore to which the control operation applies. Some operations apply to the entire set. For these operations, this argument is ignored.

The **cmd** argument specifies the operation which is to be performed. The value and type of the fourth argument (**argument**) to **semctl**, if any, is dependent on the **cmd** specification. **cmd** should be specified as one of the following symbolic values:

- **IPC_RMID** – Removes the semaphore set and its id from the system. **argument** is not used by this operation, and need not be specified.
- **IPC_SET** – Can be used to change the ownership of the semaphore set or the access rules. **argument** should be a pointer to a **struct semid_ds**, as described below. The contents of **argument->sem_perm.uid**, **argument->sem_perm.gid** and **argument->sem_perm.mode**

will be copied to the system's semaphore set id data structure.

- **IPC_STAT** – Returns the contents of the semaphore set id data structure (see below). All elements of the data structure are stored in the object addressed by **argument**, which should be a pointer to a **struct semid_ds**.
- **GETVAL** – Returns the value in the semaphore specified by the **num** argument. **argument** is not used, and may be omitted.
- **SETVAL** – Stores a specified value in the semaphore selected by the **num** argument. **argument** specifies the value to be stored, which should be of type **int**.

  Note: Successful use of the **SETVAL** operation clears any semaphore adjustment information for this semaphore in any process. (See the **semop** function description in section "semop" on page 38 for information on semaphore adjustment.) △

- **GETALL** – Stores the value of each semaphore in the set. **argument** should be a pointer to an array of unsigned shorts in which the values will be stored. The required array size can be determined by using the **IPC_STAT** command to get the number of semaphores in the set.
- **SETALL** – Stores values in all semaphores of the set. **argument** should be a pointer to an array of unsigned shorts containing the values to be stored. The required array size can be determined by using the **IPC_STAT** command.

  Note: Successful use of the **SETAlL** operation clears any semaphore adjustment information for all semaphores in the set in any process. See the **semop** function description for information on semaphore adjustment. △

- **GETNCNT** – Returns the number of callers waiting for the value of the semaphore selected by the **num** argument to become non-zero. **argument** is not used, and need not be specified.
- **GETZCNT** – Returns the number of callers waiting for the value of the semaphore selected by the **num** argument to become zero. **argument** is not used, and need not be specified.
- **GETPID** – Returns the process id of the process which most recently updated the semaphore selected by the **num** argument. **argument** is not used, and need not be specified.

Several of the **semctl** operations allow you to obtain or access the semaphore set id data structure, which is mapped by the **struct semid_ds** type defined in **sys/sem.h**.

This data structure is defined as follows:

```
struct semid_ds {
   /* permission information */
   struct ipc_perm sem_perm;

   /* number of semaphores in set */
   unsigned short sem_nsems;

   /* time of last semop call */
   time_t sem_otime;

   /* time of last change by semctl */
   time_t sem_ctime;
};
```

The **ipc_perm** structure, which contains security information about the owner and the creator of the semaphore set, is defined as follows:

```
struct ipc_perm {
   /* owner's effective user ID */
   uid_t uid;

   /* owner's effective group ID */
   gid_t gid;

   /* creator's effective user ID */
   uid_t cuid;

   /* creator's effective group ID */
   gid_t cgid;

   /* read/write permission bits */
   mode_t mode;
};
```

RETURN VALUE

For operations of **GETVAL**, **GETNCNT**, **GETZCNT** and **GETPID**, **semctl** returns the information requested. For all others, it returns **0** if successful. In all cases, it returns **-1** if unsuccessful.

USAGE NOTES

The **semctl** function can only be used with MVS 5.2.2 or a later release.

RELATED FUNCTIONS

semget, **semop**

semget

Create or find a set of semaphores

SYNOPSIS

```
#include <sys/sem.h>
int semget(key_t key, int num, int flags);
```

DESCRIPTION

The **semget** function is used to create a new set of semaphores or to locate an existing set based on a key. Semaphore sets are implemented by OpenEdition, and allow synchronization of multiple processes which do not share memory. Each semaphore in the set has an integer value, which can be raised, lowered and tested safely by multiple processes. OpenEdition semaphores allow multiple semaphores to be processed with a single call, and support both blocking and non-blocking processing. Semaphore sets, once created using **semget**,

remain in existence until explicitly destroyed with a call to **semctl**.

The **key** argument is an integral value which identifies the semaphore set desired. A **key** value of **IPC_PRIVATE** requests a new semaphore set without an associated **key**, and which can be accessed only by the queue id returned by **semget**.

The **num** argument specifies the number of semaphores in the set, if the set is created. If the semaphore set already exists, **num** must not be larger than the number of semaphores in the set.

Note: The **num** argument may be specified as zero if the set already exists, but not for a new set. △

The **flags** argument specifies zero or more option flags specifying whether or not the semaphore set already exists, and how access to the set should be regulated. The argument should be specified as **0** for no flags, or as one or more of the following symbolic constants, combined using the or operator (|):

- **IPC_CREAT** – Specifies that if a semaphore set with the requested key does not exist, it should be created. This flag is ignored if **IPC_PRIVATE** is specified.
- **IPC_EXCL** – Specifies that a semaphore set with the requested key must not already exist. This flag is ignored if **IPC_PRIVATE** is specified, or if **IPC_CREAT** is not specified.

Additionally, any of the permission bits **S_IRUSR**, **S_IWUSR**, **S_IRGRP**, **S_IWGRP**, **S_IROTH** and **S_IWOTH** may be specified, to define what users are permitted to access or modify the semaphore set. See the **umask** function description in the *SAS/C Library Reference, Volume 2*, for more information about the meaning of these flags.

RETURN VALUE

semget returns the identifier of the semaphore set if successful, or **-1** if unsuccessful.

USAGE NOTES

The **semget** function can only be used with MVS 5.2.2 or a later release.

When **semget** creates a set of semaphores, the semaphores are uninitialized. You should use the **SETALL** command of **semctl** to give the semaphores their initial values.

EXAMPLE

This example is compiled using **sascc370 -Krent -o**. This program demonstrates the use of the Open Edition IPC functions **semget()** and **semstat()**.

```
/*--------------------------------+
| POSIX/UNIX header files         |
+--------------------------------*/
#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/ipc.h>

#include <sys/sem.h>

/*--------------------------------+
| ISO/ANSI header files           |
+--------------------------------*/
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <time.h>

#include errno.h>

/*--------------------------------+
| Types                           |
+--------------------------------*/
/* argument for semctl()            */
typedef union semun
{
/* for SETVAL                       */
   int  val;

/* for IPC_STAT and IPC_SET         */
   struct semid_ds * ID;

/* for GETALL and SETALL            */
   unsigned short * array;

} SemaphoreSet;

/*--------------------------------+
| Constants                       |
+--------------------------------*/
/* semaphore key                    */
#define SEM_KEY      (key_t)1097

/* give everyone read/write         */
/* permission to semaphore          */
#define SEM_PERM  (S_IRUSR|S_IWUSR
  |S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

/*-----------------------------------+
| Name:      main                    |
| Returns:   exit(EXIT_SUCCESS) or   |
|            exit(EXIT_FAILURE)      |
+-----------------------------------*/
int main()
{
/* semaphore set id                  */
    int semID;

/* semaphore flags                   */
    int semFlags;

/* number of semaphore in a set      */
    int numSems;

/* index into semaphore set (array)  */
```

```
    int semNumber;

/* command to semaphore.              */
    int semCmd;

/* union of info on semaphore set     */
    SemaphoreSet semSet;

/*----------------------------------------*/
/* Create semaphore set with 5 semaphores. */
/* Give everyone read/write permissions.   */
/*----------------------------------------*/
/* create set with 5 semaphores    */
   numSems = 5;
   semFlags = IPC_CREAT | SEM_PERM;

if ( (semID = semget
  (SEM_KEY, numSems, semFlags)) < 0 )
   {
       perror("SEMSTAT: semget");
       exit(EXIT_FAILURE);
   }

/*-------------------------------------*/
/* Allocate memory to store            */
/* information from semaphore set      */
/*-------------------------------------*/
 semSet.ID = malloc(sizeof(struct semid_ds));

   if (semSet.ID == NULL)
   {
   fprintf(stderr,
   "ERROR:
   Cannot allocate memory for
   semaphore set stats\n");
   semctl(semID, NULL, IPC_RMID, NULL);
   exit(EXIT_FAILURE);
   }

/*-------------------------------------*/
/* Call semctl to retrieve             */
/* information from semaphore set      */
/*-------------------------------------*/
/* command to retrieve stats         */
   semCmd = IPC_STAT;
/* ignored for IPC_STAT command      */
   semNumber = 0;

if (semctl(semID, semNumber,
   semCmd, semSet) < 0)
   {
   perror("SEMSTAT: semctl failed to
   retrieve semaphore set stats");
   semctl(semID, NULL, IPC_RMID, NULL);
   free(semSet.ID);
   exit(EXIT_FAILURE);
   }

/*-------------------------------------*/
/* Print infomation retrieved from   */
/*  semaphore set                    */
/*-------------------------------------*/
 printf("Semaphore Set Statistics:\n\n");

printf("\tOwner's effective user ID: %d\n",
      (int)semSet.ID->sem_perm.uid);
printf("\tOwner's effective group ID: %d\n",
      (int)semSet.ID->sem_perm.gid);
printf("\tCreator's effective user ID: %d\n",
      (int)semSet.ID->sem_perm.cuid);
printf("\tCreator's effective group ID: %d\n",
      (int)semSet.ID->sem_perm.cgid);
printf("\tPermission mode bits: %#.3o\n\n",
      (int)semSet.ID->sem_perm.mode);

printf("\tNumber of semaphores currently in
   the set: %d\n",
      semSet.ID->sem_nsems);
printf("\tTime of last semop call: %s",
      ctime(&semSet,ID->sem_otime));
printf("\tTime of last change: %s",
      ctime(&semSet.ID->sem_ctime));

/*-------------------------------------*/
/* Free memory used to store           */
/* information from semaphore set      */
 *-------------------------------------*/
   free(semSet.ID);

/*-------------------------------------*/
/* Call semctl to remove semaphore set.  */
/*-------------------------------------*/
if (semctl(semID, NULL, IPC_RMID, NULL) < 0)
   {
   perror("SEMSTAT: semctl failed to
   remove semNumber set");
   exit(EXIT_FAILURE);
   }

   exit(EXIT_SUCCESS);

}  /* end of main() */
```

RELATED FUNCTIONS

semctl, **semop**

semop

Update semaphores atomically

SYNOPSIS

```
#include <sys/sem.h>
int semop(int id, struct sembuf *ops,
          size_t num);
```

DESCRIPTION

The **semop** function is used to update one or more semaphores from a set atomically. For each update, the caller can request either blocking until the operation can be performed, or immediate failure of **semop**.

Note: See the **semget** function description in section "semget" on page 36 for general information about OpenEdition semaphores. △

The **id** argument to **semop** specifies the id of the semaphore set to be updated. This argument is an id, such as the id returned by **semget**, not a semaphore set key, which might be passed as an argument to **semget**.

The **ops** argument should be a pointer to an array of one or more **sembuf** structures, each of which defines a single semaphore operation to be performed. All the operations are performed simultaneously and atomically; that is, no changes are made to any semaphore until they can be made to all specified semaphores. The **struct sembuf** mapping is defined by **sys/sem.h** as follows:

```
struct sembuf {
   /* semaphore number */
   unsigned short sem_num;

   /* semaphore operation code */
   short sem_op;

   /* option flags */
   short sem_flg;
};
```

The **sem_num** field of a **struct sembuf** specifies the specific semaphore to be updated. The **sem_op** field is interpreted in the following manner:

- If **sem_op** is a positive integer, the value of the indicated semaphore is increased by the specified amount.
- If **sem_op** is a negative integer and the semaphore value is greater than or equal to **-sem_op**, the semaphore value is decremented by **-sem_op**.
- If **sem_op** is a negative integer and the semaphore value is less than **-sem_op**, **semop** will either wait for the semaphore value to be raised, or return failure, depending on whether **IPC_NOWAIT** is set in **sem_flg**. If **semop** waits, once the semaphore value is greater than or equal to **-sem_op**, the value is decremented by **-sem_op**.
- If **sem_op** is zero and the semaphore value is zero, no action is taken.
- If **sem_op** is zero and the semaphore value is non-zero, **semop** will either wait for the semaphore value to become zero, or return failure, depending on whether **IPC_NOWAIT** is set in **sem_flg**.

The **sem_flg** field of the **sembuf** structure is used to specify option flags. It may contain **0** if no options are required, or one or more of the following symbolic constants, combined by the or operator (|).

- **IPC_NOWAIT** – Specifies that if the operation specified by **sem_op** cannot be immediately performed, **semop** should return failure with **errno** set to **EAGAIN** rather than waiting to perform the operation. If **IPC_NOWAIT** is not specified, **semop** will block until the operation can be performed, or until the semaphore set is destroyed or a signal is received.
- **SEM_UNDO** – Specifies that each successful semaphore operation be reflected in a semaphore adjustment value maintained for the process for each semaphore changed. When the process is terminated, each semaphore updated by the process using the **SEM_UNDO** flag is adjusted by the *semaphore adjustment* value. If **SEM_UNDO** is used consistently by the process, this will have the effect of undoing the semaphore updates performed by the process when it terminates.

The **num** argument to **semop** indicates the number of operations specified by the **ops** argument.

RETURN VALUE

semop returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **semop** function can only be used with MVS 5.2.2 or a later release.

PORTABILITY

Complex **semop** calls may be handled differently by different systems, especially if some operations specify **IPC_NOWAIT** and some do not. For best results, it is recommended that **IPC_NOWAIT** should be either specified by all elements of the **ops** array, or by none of them.

EXAMPLE

This example uses a single semaphore to synchronize access to a resource. The semaphore is created the first time the application runs, and is deleted only if the first argument to the program is the string **-d**.

```
#include <sys/sem.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

int main(int argc, char *argv[])
{
  /* semaphore set id */
  int id;

  /* inital value for semaphore set */
  unsigned short available[1] = { 1 };

  /* return code var */
  int rc;

  /* to reserve the semaphore */
  struct sembuf reserve[1];

  /* to release the semaphore */
  struct sembuf release[1];

   /* If this application's semaphore
      does not exist, create it */

  /* arg1-arbitrary unique semaphore key */
  /* arg2-only 1 semaphore in set */
```

```
/* arg3-don't create it */
id = semget(0x5A5C, 1, 0);

if (id < 0)

  /* if failed, but not because
     semaphore doesn't exist */
  if (errno != ENOENT)
  {

    badget:
    perror("semget");
    abort();
  }

  /* no semaphore, create it */
  else
  {
    /* unless its to be deleted */
    if (argc>1 &&
        strcmp(argv[1],"-d")==0)
        exit(0);
    id=semget(0x5A5C, 1,
         IPC_CREAT | S_IRUSR | S_IWUSR);
    /* if that fails, give up */
    if (id < 0) goto badget;

    /* Init semaphore to 1 (available) */
    rc=semctl(id, 0, SETALL, available);
    if (rc < 0)
    {
      perror("semctl SETALL");
      abort();
    }
  }

/* set up to reserve semaphore */
reserve[0].sem_num = 0;

/* try to decrement semaphore by 1 */
reserve[0].sem_op = -1;

/* if process dies, unreserve it */
reserve[0].sem_flg = SEM_UNDO;

/* wait for semaphore to become
   non-zero, then zero it */
rc = semop(id, reserve, 1);

if (rc < 0)
{
  perror("semop reserve");
  abort();
}

/* We now have exclusive control of the
   guarded resource. Place code here to
   update the resource. */

if (argc>1 && strcmp(argv[1],"-d") == 0)
{
    /* Check for a request to delete
       the semaphore */
    rc = semctl(id, 0, IPC_RMID);
    if (rc < 0)
    {
      perror("semctl IPC_RMID");
      abort();
    }
    exit(0);
  }

  /* Release semaphore for next request */

  /* set up to release semaphore */
  release[0].sem_num = 0;

  /* try to increment semaphore by 1 */
  release[0].sem_op = 1;

  /* if process dies, undo it */
  release[0].sem_flg = SEM_UNDO;

  rc = semop(id, release, 1);
  if (rc < 0)
  {
    perror("semop release");
      abort();
  }
  exit(0);
}
```

RELATED FUNCTIONS

semctl, **semget**

setgrent

Reposition the group database

SYNOPSIS

```
#include <sys/types.h>
#include <grp.h>
void setgrent(void);
```

DESCRIPTION

The **setgrent** function rewinds the group database so that the next call to the **getgrent** function will access the first group in the database.

RETURN VALUE

setgrent has no return value.

USAGE NOTES

The **setgrent** function can only be used with MVS 5.2.2 or a later release.

If the **setgrent** function cannot be executed (for instance, because OpenEdition is not installed or running), it issues an MVS user **ABEND 1230** to indicate the error.

RELATED FUNCTIONS

endgrent, **getgrent**

setitimer

Define an interval timer

SYNOPSIS

```
#include <sys/time.h>
int setitimer(int kind,
              const struct itimerval *ival,
              struct itimerval *oval);
```

DESCRIPTION

The **setitimer** function defines an interval timer, that is, a timer which generates a signal each time a specified time interval expires. Three different forms of time measurement may be specified.

The **kind** argument is a symbolic constant specifying the type of time interval required. The permitted values are:

- **ITIMER_REAL** – Specifies a real time interval. Each time the interval expires, a **SIGALRM** signal is generated.

  Note: A call to the **alarm** function will cancel a real time interval defined by **setitimer**, and similarly, a call to **setitimer** specifying **ITIMER_REAL** will cancel an outstanding **alarm**. △

- **ITIMER_VIRTUAL** – Specifies a virtual (CPU) time interval. Each time the interval expires, a **SIGVTALRM** signal is generated.
- **ITIMER_PROF** – Specifies a profiling interval, measuring CPU time plus system time used in behalf of this process. Each time the interval expires, a **SIGPROF** signal is generated.

The **ival** argument specifies the time values controlling the timer. This argument is a pointer to an **itimerval** structure, which contains two fields defined as follows:

- **it_value** – the time until the next expiration of the timer
- **it_interval** – the time between successive timer expirations.

In other words, the **it_value** field specifies the amount of time between the **setitimer** call and the first expiration, while the **it_interval** field specifies the time between successive expirations. If **it_value** is specified as zero when **setitimer** is called, any existing interval is immediately cancelled. If **it_interval** is specified as zero, the timer will expire once, and then be cancelled.

The **it_value** and **it_interval** fields both have type **struct timeval**, which allows a time value to be specified to an accuracy of a microsecond. The structure has two fields:

- **tv_sec** – the number of seconds in the interval
- **tv_usec** – the number of microseconds to be added to the interval.

The **oval** argument to **setitimer** is a pointer to a **struct itimerval** in which the current interval timer values are to be stored. **oval** may be specified as **NULL**, in which case, this information is not stored.

RETURN VALUE

setitimer returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **setitimer** function can only be used with MVS 5.2.2 or a later release.

Programs which are not invoked from the shell must call the **oesigsetup** function to enable handling of the timer signals before invoking **setitimer**.

RELATED FUNCTIONS

getitimer

setpriority

Change process priority

SYNOPSIS

```
#include <sys/resource.h>
int setpriority(int kind, int id,
                int prio);
```

DESCRIPTION

The **setpriority** function changes the OpenEdition priority of a process, or the priority of all processes in a process group or belonging to a user. See the **getpriority** function description in section “getpriority” on page 16 for further information on OpenEdition priorities.

The **kind** argument to **setpriority** should be specified as a symbolic constant indicating the scope of the priority change. The permissible values are:

- **PRIO_PROCESS** – specifies that the **id** argument is the pid of the process whose priority is to be changed
- **PRIO_PGRP** – specifies that the **id** argument is the pid of the process group whose processes should be changed in priority
- **PRIO_USER** – specifies that the **id** argument is the uid of the user whose processes are to be changed in priority.

The **id** argument specifies the process id, process group id, or user id whose priority should be changed. If **id** is **0**, it specifies the calling process, process group or user.

The **prio** argument specifies the requested new priority. It should be a signed integer between -20 to 19. Lower numbers indicate higher priority.

Note: OpenEdition sites must enable the use of the **setpriority** function. If the use of **setpriority** has not been enabled, any use of **setpriority** will fail with **errno** set to **ENOSYS**. △

RETURN VALUE

setpriority returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **setpriority** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE

Refer to "chpriority" on page 8 for an example that demonstrates the use of the OpenEdition process priority functions **chpriority**, **getpriority**, and **setpriority**.

RELATED FUNCTIONS

chpriority, **getpriority**

setpwent

Reposition the user database

SYNOPSIS

```
#include <sys/types.h>
#include <pwd.h>
void setpwent(void);
```

DESCRIPTION

The **setpwent** function rewinds the user database so that the next call to the **getpwent** function will access the first user in the database.

RETURN VALUE

setpwent has no return value.

USAGE NOTES

The **setpwent** function can only be used with MVS 5.2.2 or a later release.

If the **setpwent** function cannot be executed (for instance, because OpenEdition is not installed or running), it issues an MVS user **ABEND 1230** to indicate the error.

RELATED FUNCTIONS

endpwent, **getpwent**

setregid

Set real and/or effective group id

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
int setregid(gid_t realgid,
             gid_t effgid);
```

DESCRIPTION

The **setregid** function is used to set the real and/or the effective group ids for the calling process. A superuser or daemon process has the ability to set any valid group id. Other processes are limited in their use of **setregid**.

Note: See the *IBM OpenEdition MVS Assembler Callable Services* publication, SC28-2899, for more information on the use of **setregid** by unprivileged processes. △

The **realgid** argument to **setregid** specifies the new real group id. If **realgid** is specified as **-1**, the real group id is unchanged. The **effgid** argument to **setregid** specifies the new effective group id. If **effgid** is specified as **-1**, the effective group id is unchanged.

RETURN VALUE

setregid returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **setregid** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE

This example is compiled using **sascc370 -Krent -o**. This program demonstrates the use of the OpenEdition functions **setregid()** and **setreuid()**.

```
/*--------------------------------------+
| POSIX/UNIX header files               |
+--------------------------------------*/
#include <sys/types.h>

#include <unistd.h>

/*--------------------------------------+
| ISO/ANSI header files                 |
+--------------------------------------*/
#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

/*--------------------------------------+
| Name:      main                       |
| Returns:   exit(EXIT_SUCCESS) or      |
|            exit(EXIT_FAILURE)         |
+--------------------------------------*/
int main()
{
/* generic return code                          */
   int rc;

/* real group ID of calling process             */
   gid_t rgid;

/* effective group ID of calling process        */
   gid_t egid;

/* real user ID of calling process              */
   uid_t ruid;

/* effective user ID of calling process         */
   uid_t euid;

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

```
/* Get the real and effective group and user    */
/* ids for this process.                        */
/*----------------------------------------------*/
   printf("\nGet the Real and Effective
      Group and User IDs\n");
/* get real group ID                            */
   rgid = getgid();

/* get effective group ID                       */
   egid = getegid();

/* get real user ID                             */
   ruid = getuid();

/* get effective user ID                        */
   euid = geteuid();

   printf("     The real group id is: %d\n",
    (int)rgid);
   printf("The effective group id is: %d\n",
    (int)egid);
   printf("      The real user id is: %d\n",
    (int)ruid);
   printf(" The effective user id is: %d\n",
    (int)euid);

/*----------------------------------------------*/
/* Set the real and effective group id for this  */
/* process.                                      */
/*----------------------------------------------*/
printf("\nSetting the Real and Effective Group ID\n");

   errno = 0;
   egid = rgid;
/* -1 implies use current real group id (nochg) */
   rgid = (gid_t)-1;

   rc = setregid(rgid, egid);

   /* Test for Error */
   if (rc == -1)
   {
      perror("Call to setregid failed");
      exit(EXIT_FAILURE);
   }

/*----------------------------------------------*/
/* Set the real and effective user id for     */
/* this process.                                */
/*----------------------------------------------*/
 printf("\nSetting the Real and
   Effective User ID\n");

   errno = 0;
   euid = ruid;
/* -1 implies use current real user id (nochg) */
   ruid = (gid_t)-1;

   rc = setreuid(ruid, euid);

   /* Test for Error */
      if (rc == -1)
      {
         perror("Call to setreuid failed");
         exit(EXIT_FAILURE);
      }

   /*---------------------------------------*/
   /* Get the real and effective group and    */
   /* user ids                              */
   /*---------------------------------------*/
      printf("\nThe Real and Effective
        Group and User IDs are now!\n");

   /* get real group ID                     */
      rgid = getgid();

   /* get effective group ID                */
      egid = getegid();

   /* get real user ID                      */
      ruid = getuid();

   /* get effective user ID                 */
      euid = geteuid();

      printf("     The real group id is: %d\n",
       (int)rgid);
      printf("The effective group id is: %d\n",
       (int)egid);
      printf("      The real user id is: %d\n",
       (int)ruid);
      printf(" The effective user id is: %d\n",
       (int)euid);

      exit(EXIT_SUCCESS);

   }  /* end of main() */
```

RELATED FUNCTIONS

getegid, **getgid**, **setegid**, **setgid**

setreuid

Set real and/or effective user id

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
int setreuid(uid_t realuid, uid_t effuid);
```

DESCRIPTION

The **setreuid** function is used to set the real and/or the effective user ids for the calling process. A superuser or daemon process has the ability to set any valid user id. Other processes are limited in their use of **setreuid**.

Note: See the *IBM OpenEdition MVS Assembler Callable Services* publication, SC28-2899, for more information on the use of **setreuid** by unprivileged processes. △

The **realuid** argument to **setreuid** specifies the new real user id. If **realuid** is specified as **-1**, the real user id is unchanged. The **effuid** argument to **setreuid** specifies the new effective user id. If **effuid** is specified as **-1**, the effective user id is unchanged.

RETURN VALUE

setreuid returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **setreuid** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE

Refer to "setregid" on page 42 for an example that demonstrates the use of the OpenEdition functions **setregid()** and **setreuid()**.

RELATED FUNCTIONS

geteuid, **getuid**, **seteuid**, **setuid**

setrlimit

Define OpenEdition resource limits

SYNOPSIS

```
#include <sys/resource.h>
int setrlimit(int resource,
              const struct rlimit *info);
```

DESCRIPTION

The **setrlimit** function defines resource limits for the calling process. Limits are expressed as a pair of integers, a soft limit and a hard limit. The soft limit controls the amount of the resource the program is actually allowed to consume, while the hard limit specifies an upper bound on the soft limit. The soft limit can be raised up to the hard limit value, but the hard limit can only be raised by a privileged (superuser) caller.

The **resource** argument defines the resource to which the limit applies. It should be specified as one of the symbolic values defined below.

- **RLIMIT_AS** – Specifies the maximum size in bytes of the address space for this process. If the limit is exceeded, memory allocation functions such as **malloc** or **GETMAIN** will be unable to allocate additional memory.

  Note: Setting this limit too low could cause the program to be terminated due to the inability of the library to obtain more stack space. △

- **RLIMIT_CORE** – Specifies the maximum size in bytes of an OpenEdition memory dump (core file) for this process. A limit of **0** will prevent a dump file from being created.
- **RLIMIT_CPU** – Specifies the maximum CPU time in seconds allowed for this address space. When the limit is exceeded, a **SIGXCPU** signal is sent to the process, and a small amount of additional time is allocated to allow a signal handler to execute.
- **RLIMIT_DATA** – Specifies the maximum amount of memory available for data allocation using **malloc** or **calloc**. This limit is not enforced under OpenEdition MVS, and an attempt to set the limit lower than **RLIM_INFINITY** will be rejected.
- **RLIMIT_FSIZE** – Specifies the maximum file size in bytes allowed for this process. A limit of **0** will prevent new files from being created. If an attempt is made to extend a file beyond the limit, a **SIGXFSZ** signal is sent to the process. The limit applies only to OpenEdition HFS files, not to standard MVS data sets.
- **RLIMIT_NOFILE** – The maximum number of file descriptors the process may have open. Any attempt to open a file with a file number above the limit will fail with **errno** set to **EMFILE**.
- **RLIMIT_STACK** – The maximum amount of memory available for stack allocation. This limit is not enforced under OpenEdition MVS, and an attempt to set the limit lower than **RLIM_INFINITY** will be rejected.

The **info** argument is a pointer to a structure of type **struct rlimit**, which defines the soft and hard limits. The structure contains the following fields:

1. **rlim_cur** – the current (i.e., soft) limit
2. **rlim_max** – the maximum limit

RETURN VALUE

setrlimit returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **setrlimit** function can only be used with MVS 5.2.2 or a later release.

Note: Resource limits defined by **setrlimit** are propagated to child processes created by **fork** or **exec**. △

RELATED FUNCTIONS

getrlimit

shmat

Attach a shared memory segment

SYNOPSIS

```
#include <sys/shm.h>
void *shmat(int id, const void *addr,
            int flags);
```

DESCRIPTION

The **shmat** function is used to attach a shared memory segment to a process, so that the memory contents can be accessed.

Note: See the **shmget** function description in section "shmget" on page 48 for general information about shared memory segments. △

The **id** argument to **shmat** specifies a shared memory segment id. This argument is an id, such as the id returned by **shmget**, not a memory segment key, which might be passed as an argument to **shmget**.

The **addr** argument to **shmat** specifies a pointer value indicating the address at which the memory segment is to be attached. If **addr** is **NULL**, the segment will be attached at an address selected by the system. If **addr** is specified, **shmat** will fail if the segment cannot be attached as specified because memory is already allocated near the address specified.

Note: The flag bit **SHM_RND** influences the interpretation of **addr**, as described below. △

The **flags** argument specifies zero or more option flags. The argument should be specified as **0** for no flags, or as one or more of the following symbolic constants, combined using the or operator (|):

- **SHM_RDONLY** – Specifies that the segment is attached read-only, that is, the process will be able to read the memory contents, but not change them.
- **SHM_RND** – Specifies that the address specified by **addr** will be rounded down to a multiple of the page size. The page size is defined by the symbolic constant **SHMLBA**.

RETURN VALUE

shmat returns the address of the attached segment, or (void *) **-1** if unsuccessful.

USAGE NOTES

The **shmat** function can only be used with MVS 5.2.2 or a later release.

Note: A site can impose limits on the size and number of shared memory segments that can be attached. △

EXAMPLE 1

This example is compiled using **sascc370 -Krent -o**. This program uses the functions **shmat()**, **shmctl()**, **shmdt()**, and **shmget()** to establish an IPC Client using a Shared Memory Segment.

```
/*-------------------------------------+
| POSIX/UNIX header files              |
+-------------------------------------*/
#include <fcntl.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include sys/shm.h>

/*-------------------------------------+
| ISO/ANSI header files                |
+-------------------------------------*/
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

/*-------------------------------------+
| Constants                            |
+-------------------------------------*/
/* memory segment character value     */
#define MEM_CHK_CHAR        '*'

/* shared memory key                  */
#define SHM_KEY      (key_t)1097

#define SHM_SIZE     (size_t)256

/* size of memory segment (bytes)     */
/* give everyone read/write           */
/* permission to shared memory        */
#define SHM_PERM  (S_IRUSR|S_IWUSR
  |S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

/*-------------------------------------+
| Name:      main                      |
| Returns:   exit(EXIT_SUCCESS) or     |
|            exit(EXIT_FAILURE)        |
+-------------------------------------*/
int main()
{
/* loop counter                       */
    int i;

/* shared memory segment id           */
    int shMemSegID;

/* shared memory flags                */
    int shmFlags;

/* ptr to shared memory segment       */
    char * shMemSeg;

/* generic char pointer               */
    char * cptr;

/*-------------------------------------*/
/* Get the shared memory segment for   */
/* SHM_KEY, which was set by           */
/* the shared memory server.           */
/*-------------------------------------*/
   shmFlags = SHM_PERM;

if ( (shMemSegID =
shmget(SHM_KEY, SHM_SIZE, shmFlags)) < 0 )
   {
   perror("CLIENT: shmget");
   exit(EXIT_FAILURE);
   }

/*-----------------------------------------*/
/* Attach the segment to the process's     */
/* data space at an address                */
/* selected by the system.                 */
/*-----------------------------------------*/
shmFlags = 0;
```

```
if ( (shMemSeg =
      shmat(shMemSegID, NULL, shmFlags)) ==
      (void *) -1 )
   {
       perror("SERVER: shmat");
       exit(EXIT_FAILURE);
   }

/*-----------------------------------------*/
/* Read the memory segment and verify that   */
/* it contains the values                    */
/* MEM_CHK_CHAR and print them to the screen */
/*-----------------------------------------*/
for (i=0, cptr = shMemSeg; i < SHM_SIZE;
i++, cptr++)
   {
      if ( *cptr != MEM_CHK_CHAR )
      {
         fprintf(stderr, "CLIENT:
           Memory Segment corrupted!\n");
         exit(EXIT_FAILURE);
      }

      putchar( *cptr );
/* print 40 columns across            */

      if ( ((i+1) % 40) == 0 )
     {
         putchar('\n');
      }
   }
   putchar('\n');

/*-----------------------------------------*/
/* Clear shared memory segment.              */
/*-----------------------------------------*/
   memset(shMemSeg, '\0', SHM_SIZE);

/*-----------------------------------------*/
/* Call shmdt() to detach shared           */
/* memory segment.                         */
/*-----------------------------------------*/
   if ( shmdt(shMemSeg) < 0 )
   {
       perror("SERVER: shmdt");
       exit(EXIT_FAILURE);
   }

   exit(EXIT_SUCCESS);

}  /* end of main() */
```

EXAMPLE 2

This example is compiled using **sascc370 -Krent -o**. This program uses the functions **shmat()**, **shmctl()**, **shmdt()**, and **shmget()** to establish an IPC Server using a Shared Memory Segment.

Note: One cannot use the Extended Message structure to send a message, i.e., call **shmsnd()** with this structure. Attempting to do so will cause the program to "hang". △

```
/*--------------------------------------+
| POSIX/UNIX header files               |
+--------------------------------------*/
#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/ipc.h>

#include >sys/shm.h>

/*--------------------------------------+
| ISO/ANSI header files                 |
+--------------------------------------*/
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <time.h>

#include <errno.h>

/*--------------------------------------+
| Constants                             |
+--------------------------------------*/
/* value to fill memory segment        */
#define MEM_CHK_CHAR         '*'

/* shared memory key                   */
#define SHM_KEY      (key_t)1097

/* size of memory segment (bytes)      */
#define SHM_SIZE     (size_t)256

/* give everyone read/write permission   */
/* to shared memory                      */
#define SHM_PERM  (S_IRUSR|S_IWUSR
  |S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

/*--------------------------------------+
| Name:      main                       |
| Returns:   exit(EXIT_SUCCESS) or      |
|            exit(EXIT_FAILURE)         |
+--------------------------------------*/
int main()
{
/* shared memory segment id           */
    int shMemSegID;

/* shared memory flags                */
    int shmFlags;

/* ptr to shared memory segment       */
    char * shMemSeg;

/*-----------------------------------------*/
/* Create shared memory segment          */
```

```
/* Give everyone read/write permissions. */
/*---------------------------------------*/
   shmFlags = IPC_CREAT | SHM_PERM;

 if ( (shMemSegID =
  shmget(SHM_KEY, SHM_SIZE, shmFlags)) < 0 )
   {
       perror("SERVER: shmget");
       exit(EXIT_FAILURE);
   }

/*-----------------------------------------*/
/* Attach the segment to the process's data  */
/* space at an address                       */
/* selected by the system.                   */
/*-----------------------------------------*/
shmFlags = 0;
if ( (shMemSeg =
      shmat(shMemSegID, NULL, shmFlags)) ==
  (void *) -1 )
   {
       perror("SERVER: shmat");
       exit(EXIT_FAILURE);
   }

/*-----------------------------------------*/
/* Fill the memory segment with MEM_CHK_CHAR */
/* for other processes to read               */
/*-----------------------------------------*/
   memset(shMemSeg, MEM_CHK_CHAR, SHM_SIZE);

/*-------------------------------------------------*/
/* Go to sleep until some other process changes  */
/* first character                               */
/* in the shared memory segment.                 */
/*-------------------------------------------------*/
   while (*shMemSeg == MEM_CHK_CHAR)
   {
      sleep(1);
   }

/*-------------------------------------------------*/
/* Call shmdt() to detach shared memory segment.  */
/*-------------------------------------------------*/
   if ( shmdt(shMemSeg) < 0 )
   {
       perror("SERVER: shmdt");
       exit(EXIT_FAILURE);
   }


/*-------------------------------------------------*/
/* Call shmctl to remove shared memory segment.     */
/*-------------------------------------------------*/
   if (shmctl(shMemSegID, IPC_RMID, NULL) < 0)
   {
       perror("SERVER: shmctl");
       exit(EXIT_FAILURE);
   }

   exit(EXIT_SUCCESS);
}  /* end of main() */
```

RELATED FUNCTIONS

shmctl, **shmdt**, **shmget**

shmctl

Control a shared memory segment

SYNOPSIS

```
#include <sys/shm.h>
int shmctl(int id, int cmd,
           struct shmid_ds *buf);
```

DESCRIPTION

The **shmctl** function is used to perform one of several control operations on a shared memory segment.

Note: See the **shmget** function description in section "shmget" on page 48 for general information about shared memory segments. △

The **id** argument to **shmctl** specifies a shared memory segment id. This argument is an id, such as the id returned by **shmget**, not a memory segment key, which might be passed as an argument to **shmget**.

The **cmd** argument should be specified as a symbolic constant specifying the specific operation to be performed by **shmctl**. The constant values are described below.

Several of the **shmctl** operations allow you to obtain or access the shared memory id data structure, which is mapped by the **struct shmid_ds** type defined in **sys/shm.h**.

This data structure is defined as follows:

```
struct shmid_ds {
   /* permission information */
   struct ipc_perm shm_perm;

   /* segment size */
   int shm_segsz;

   /* process id for last shm operation */
   pid_t shm_lpid;

   /* process id of creator */
   pid_t shm_cpid;

   /* number of times segment attached */
   unsigned shm_nattch;

   /* time of last shmat call */
   time_t shm_atime;

   /* time of last shmdt call */
   time_t shm_dtime;

   /* time of last change by shmget/shmctl */
   time_t shm_ctime;
```

```
};
```

The **ipc_perm** structure contains security information about the owner and the creator of the memory segment and is defined as follows:

```
struct ipc_perm {
   /* owner's effective user ID */
   uid_t uid;

   /* owner's effective group ID */
   gid_t gid;

   /* creator's effective user ID */
   uid_t cuid;

   /* creator's effective group ID */
   gid_t cgid;

   /* read/write permission bits */
   mode_t mode;
};
```

For **shmctl** operations which access or modify the shared memory id data structure, the **buf** argument addresses a **struct shmid_ds**, used as described below. For other operations, the **buf** argument is ignored.

The **cmd** values accepted by **shmctl** and their meanings are as follows:

- **IPC_RMID** – Removes the shared memory segment and its id from the system, after all users have detached it. The **buf** argument is not used by this operation.
- **IPC_SET** – Can be used to change the ownership of a shared memory segment or the access rules. The contents of **buf->shm_perm.uid**, **buf->shm_perm.gid** and **buf->shm_perm.mode** will be copied to the shared memory id data structure.
- **IPC_STAT** – Returns the contents of the shared memory id data structure. All elements of the data structure are stored in the object addressed by **buf**.

RETURN VALUE

shmctl returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **shmctl** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE 1

Refer to "shmat" on page 44 for an example that uses the functions **shmat()**, **shmctl()**, **shmdt()**, and **shmget()** to establish an IPC Client using a Shared Memory Segment.

EXAMPLE 2

Refer to "shmat" on page 44 for an example that uses the functions **shmat()**, **shmctl()**, **shmdt()**, and **shmget()** to establish an IPC Server using a Shared Memory Segment.

RELATED FUNCTIONS

shmat, **shmdt**, **shmget**

shmdt

Detach a shared memory segment

SYNOPSIS

```
#include <sys/shm.h>
int shmdt(const void *addr);
```

DESCRIPTION

The **shmdt** function is used to detach a shared memory segment from a process. The segment is not destroyed, even if the calling process is the only process which has it attached. (See the **shmget** function description in section "shmget" on page 48 for general information about shared memory segments.)

The **addr** argument specifies a pointer value to the location at which the shared segment is attached.

RETURN VALUE

shmdt returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES

The **shmdt** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE 1

Refer to "shmat" on page 44 for an example that uses the functions **shmat()**, **shmctl()**, **shmdt()**, and **shmget()** to establish an IPC Client using a Shared Memory Segment.

EXAMPLE 2

Refer to "shmat" on page 44 for an example that uses the functions **shmat()**, **shmctl()**, **shmdt()**, and **shmget()** to establish an IPC Server using a Shared Memory Segment.

RELATED FUNCTIONS

shmat, **shmctl**, **shmget**

shmget

Create or find a shared memory segment

SYNOPSIS

```
#include <sys/shm.h>
int shmget(key_t key, size_t size,
           int flags);
```

DESCRIPTION

The **shmget** function is used to create a new shared memory segment or to locate an existing one based on a key. Shared memory segments are memory areas which can be shared by several processes and which, once created, continue to exist until explicitly deleted using the **shmctl** function.

The **key** argument is an integral value which identifies the shared memory segment desired. A **key** value of

IPC_PRIVATE requests a new shared memory segment without an associated key, and which can be accessed only by the segment id returned by **shmget**.

The **size** argument specifies the required size of the segment. If the segment already exists, **size** must be no greater than the size specified when the segment was created.

The **flags** argument specifies zero or more option flags specifying whether or not the segment already exists, and how access to the segment should be regulated. The argument should be specified as **0** for no flags, or as one or more of the following symbolic constants, combined using the or operator (|):

- **IPC_CREAT** – Specifies that if a segment with the requested key does not exist, it should be created. This flag is ignored if **IPC_PRIVATE** is specified.
- **IPC_EXCL** – Specifies that a segment with the requested key must not already exist. This flag is ignored if **IPC_PRIVATE** is specified, or if **IPC_CREAT** is not specified.

Additionally, any of the permission bits **S_IRUSR**, **S_IWUSR**, **S_IRGRP**, **S_IWGRP**, **S_IROTH** and **S_IWOTH** may be specified, to define what users are permitted to access or modify the memory segment. See the **umask** function description in the *SAS/C Library Reference, Volume 2*, for more information about the meaning of these flags.

RETURN VALUE

shmget returns the identifier of the shared memory segment if successful, or **-1** if unsuccessful.

USAGE NOTES

The **shmget** function can only be used with MVS 5.2.2 or a later release.

Note: A site can impose limits on the size and number of shared memory segments created. △

EXAMPLE 1

Refer to "shmat" on page 44 for an example that uses the functions **shmat()**, **shmctl()**, **shmdt()**, and **shmget()** to establish an IPC Client using a Shared Memory Segment.

EXAMPLE 2

Refer to "shmat" on page 44 for an example that uses the functions **shmat()**, **shmctl()**, **shmdt()**, and **shmget()** to establish an IPC Server using a Shared Memory Segment.

EXAMPLE 3

This example is compiled using **sascc370 -Krent -o**. This program demonstrates the functions **shmget()** and **shmstat()**. It uses the **shmget()** function to create IPC shared memory segment and then uses the **shmctl()** function to retrieve statistics on the newly created memory segment.

```
/*-------------------------------------+
| POSIX/UNIX header files              |
+-------------------------------------*/
#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/ipc.h>

#include <sys/shm.h>

/*-------------------------------------+
| ISO/ANSI header files                |
+-------------------------------------*/
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <time.h>

#include <errno.h

/*-------------------------------------+
| Types                                |
+-------------------------------------*/
/* shared memory segment structure */
typedef struct shmid_ds ShMemSeg;

/*-------------------------------------+
| Constants                            |
+-------------------------------------*/
/* shared memory key                   */
#define SHM_KEY      (key_t)1097

/* size of memory segment (bytes)      */
#define SHM_SIZE     (size_t)256

/* give everyone read/write             */
/* permission to shared memory          */
#define SHM_PERM  (S_IRUSR|S_IWUSR
 |S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

/*-------------------------------------+
| Name:      main                      |
| Returns:   exit(EXIT_SUCCESS) or     |
|            exit(EXIT_FAILURE)        |
+-------------------------------------*/
int main()
{
/* shared memory segment id            */
    int shMemSegID;

/* shared memory flags                 */
    int shmFlags;

/* command to shared memory            */
    int shmCmd;

/* ptr to shared mem id structure      */
    ShMemSeg * shMemSeg;
```

```
/*--------------------------------------*/
/* Create shared memory segment         */
/* Give everyone read/write permissions */
/*--------------------------------------*/
   shmFlags = IPC_CREAT | SHM_PERM;

if ( (shMemSegID =
shmget(SHM_KEY, SHM_SIZE, shmFlags)) < 0 )
   {
       perror("SHMSTAT: shmget");
       exit(EXIT_FAILURE);
   }

/*-------------------------------------------*/
/* Allocate memory to store information      */
/* from shared memory seg                    */
/*-------------------------------------------*/
   shMemSeg = malloc(sizeof(ShMemSeg));

   if (shMemSeg == NULL)
   {
   fprintf(stderr,
   "ERROR:
   Cannot allocate memory for
   shared mem stats\n");
   exit(EXIT_FAILURE);
   }

/*-------------------------------------------*/
/* Call shmctl to retrieve information from  */
/* shared memory seg                         */
/*-------------------------------------------*/
   shmCmd = IPC_STAT;

   if (shmctl(shMemSegID, shmCmd, shMemSeg) < 0)
   {
   perror("SHMSTAT: shmctl failed to
   retrieve shared mem stats");
   free(shMemSeg);
   exit(EXIT_FAILURE);
   }

/*-------------------------------------------*/
/* Print infomation retrieved from shared    */
/* memory segment                            */
/*-------------------------------------------*/
 printf("Shared Memory Segment statistics:\n\n");

 printf("\tSegment owner's effective user ID:
   %d\n",(int)shMemSeg->shm_perm.uid);
 printf("\tSegment owner's effective group ID:
   %d\n",(int)shMemSeg->shm_perm.gid);
 printf("\tSegment creator's effective user ID:
   %d\n",(int)shMemSeg->shm_perm.cuid);
 printf("\tSegment creator's effective group ID:
   %d\n",(int)shMemSeg->shm_perm.cgid);
 printf("\tSegment permission mode bits:
   %#.3o\n\n",(int)shMemSeg->shm_perm.mode);

 printf("\tShared Memory Segment size:
   %d\n",shMemSeg->shm_segsz);
 printf("\tProcess id of last Segment operation:
   %d\n",(int)shMemSeg->shm_lpid);
 printf("\tProcess id of Segment creator:
   %d\n",(int)shMemSeg->shm_cpid);
 printf("\tNumber of times Segment attached:
   %u\n",shMemSeg->shm_nattch);
 printf("\tTime of last shmat call:
   %s", ctime(&shMemSeg->shm_atime));
 printf("\tTime of last shmdt call:
   %s", ctime(&shMemSeg->shm_dtime));
   printf("\tTime of last change:
   %s", ctime(&shMemSeg->shm_ctime));

/*-------------------------------------------*/
/* Free memory used to store information     */
/* from shared memory seg                    */
/*-------------------------------------------*/
   free(shMemSeg);

/*-------------------------------------------*/
/* Call shmctl to remove shared memory segment */
/*-------------------------------------------*/
   shmCmd = IPC_RMID;

 if (shmctl(shMemSegID, shmCmd, NULL) < 0)
 {
    perror("SHMSTAT:
    shmctl failed to remove shared mem");
    exit(EXIT_FAILURE);
   }

   exit(EXIT_SUCCESS);

}  /* end of main() */
```

RELATED FUNCTIONS

shmat, **shmctl**, **shmdt**

sigblkjmp

Intercept longjmp without changing signal mask

SYNOPSIS

```
#include <lcjmp.h>
int sigblkjmp(sigjmp_buf env);
```

DESCRIPTION

sigblkjmp requests interception of calls to **longjmp** or **siglongjmp** that could terminate the calling function. When you call **sigblkjmp**, it always returns **0**. If a call to **longjmp** or **siglongjmp** is later intercepted, the call to **sigblkjmp** is resumed and upon completion returns the integer argument that was passed to **longjmp**. The **env** variable is modified to indicate the target of the intercepted call so it can be resumed by a call to **siglongjmp**.

Note: When a **siglongjmp** call is intercepted due to the use of **sigblkjmp**, the signal mask has not yet been changed. △

After a call to **longjmp** or **siglongjmp** is intercepted, **sigblkjmp** must be re-issued if continued interception is wanted.

Because **exit** is implemented as a **longjmp** to the caller of **main**, you can use **sigblkjmp** to intercept program exit.

RETURN VALUE

sigblkjmp normally returns **0**; it returns a non-zero value if a call to **longjmp** or **siglongjmp** has been intercepted (in which case **sigblkjmp** returns the value of the second argument passed to **longjmp** or **siglongjmp**).

CAUTION

Variables of storage **class auto** and **register** whose values are changed between the **sigblkjmp** and **siglongjmp** calls have indeterminate values on return to **sigblkjmp**.

EXAMPLE

This example demonstrates how **sigblkjmp** can be used to enable a function to release resources even if terminated by a call to **longjmp** or **siglongjmp** in a function that **sigblkjmp** calls:

```
#include <stdio.h>
#include <lcjmp.h>
#include <stdlib.h>
#include <lcsignal.h>

sigjmp_buf env;

static void get_resource(void),
  use_resource(void);

int main()
{
  int code;
  if (code = sigsetjmp(env,1))
    goto escape;

  get_resource();
  puts("get_resource returned normally.");
  exit(0);

 escape:
  printf("Executing escape routine for ",
         "error %d\n", code);
  exit(code);
}

static void get_resource(void)
{
  int code;
  sigjmp_buf my_env;
  sigset_t blockall, oldset;

  sigfillset(&blockall);
  /* block all signals while allocating
     and using resource */
  sigprocmask(SIG_SETMASK, &blockall,
              &oldset);

  /* Allocate resource here */
  if (code = sigblkjmp(my_env))
    goto release;

  puts("Resources allocated.");
  /* Free resource here */
  use_resource();
  puts("use_resource returned normally, "
    "get_resource is freeing resources.");
  setprocmask(SIG_SETMASK, &oldset, NULL);
  return;
 release:

  printf("use_resource indicated ",
         "error %d\n", code);
  puts("Resources now freed, proceeding ",
       "with longjmp.");
  siglongjmp(my_env, code);
}

static void use_resource(void)
{
  puts("Entering use_resource.");
  /* Attempt to use resource here. */
  puts("Error 3 detected, ",
       "calling siglongjmp.");
  siglongjmp(env, 3);
  puts("This statement will not ",
       "be executed.");
}
```

RELATED FUNCTIONS

blkjmp, **longjmp**, **setjmp**, **siglongjmp**, **sigsetjmp**

spawn

Spawn a new process

SYNOPSIS

```
#include <spawn.h>
pid_t spawn(const char *path,
            int count,
            const int fd_map[],
            struct inheritance *inh,
            const char *argv[],
            const char *envp[]);
```

DESCRIPTION

The **spawn** function creates a new process to run an executable program in the hierarchical file system. If indicated by an environment variable, an attempt is made to execute the process in the same address space as the caller. **spawn** permits the calling process to remap file descriptors, alter the signal handling, and change the process group of the new process.

The **path** argument specifies the name of the HFS file to be executed.

The **count** specifies the number of file descriptors to be remapped, and **fd_map** specifies a list of file descriptors that describe the remapping. If **fd_map** is **0**, no remapping is performed, and the child process receives all file descriptors open in the current process. If **fd_map** is not **0**, then for *n* less than **count**, file descriptor *n* in the child process is mapped to be the same file as **fd_map[n]** in the parent process. If **fd_map[n]** has the value **SPAWN_FDCLOSED**, file descriptor *n* will be closed in the child process. All file descriptors greater than or equal to **count** will be closed in the child process.

The **inh** argument specifies a pointer to an inheritance structure which defines how signals and process groups should be handled in the child process. The inheritance structure contains the following fields:

flags	bit flags defining required inheritance options
pgroup	an alternate process group for the new process
sigmask	a new signal mask for the child process
sigdefault	a set of signals to restore to default handling
ctlttyfd	a controlling terminal file descriptor for the child process

The **inheritance** structure can be used to request the following inheritance options.

If the **SPAWN_SETGROUP** flag is set in **inh->flags**, the child's process group will be set as specified by **inh->pgroup**. If **inh->pgroup** is **0**, the child will be created in a new process group. Otherwise, the new process will be assigned to the specified process group. If **SPAWN_SETGROUP** is not set, the new process is part of the same process group as the parent.

If the **SPAWN_SETSIGDEF** flag is set in **inh->flags**, each signal specified in **inh->sigdefault** (a value of type **sigset_t**) is reset to default handling in the child process. All other signals inherit their handling from the parent process, as with the **exec** functions.

If the **SPAWN_SETSIGDEF** flag is not set, all signals inherit their handling from the parent in the manner of **exec**. If the **SPAWN_SETSIGMASK** flag is set in **inh->flags**, the initial signal mask for the new child process is as specified by **inh->sigmask** (a value of type **sigset_t**). If the **SPAWN_SETSIGMASK** flag is not set, the child process inherits the signal mask of the parent process.

If the **SPAWN_SETTCPGRP** flag is set in **inh->flags**, the file descriptor specified by **inh->ctlttyfd** becomes the controlling terminal for the child process's foreground process group. If the **SPAWN_SETTCPGRP** flag is not set, the child process inherits the controlling terminal file descriptor from the parent process.

The **argv** argument to **spawn** specifies a list of arguments to be passed to the new process. **argv** is an array of strings, where **argv[0]** contains the name of the executable file, and the final element of **argv** is a **NULL** pointer value.

The **envp** argument to **spawn** specifies an array of environment variable values. Each element of the array is a string of the form:

var=value

The last element of the array must be a **NULL** pointer to indicate the end of the array. If the new process should inherit the environment variables of the calling process, pass the external environment variable pointer **environ** as **envp**.

Certain environment variables may be defined in the **envp** array to modify the operation of **spawn**. These are as follows:

If the environment variable **_BPX_SHAREAS** is defined and has the value **YES**, **spawn** will attempt to create the new process in the same address space as the parent address space.

Note: The **spawn** may be unable to use the same address space for security or performance reasons, in which case a new address space will be created for the process. △

Note: When several processes run in a single address space, some requirements of the

```
posix
```

standards are violated. (For instance, it is not possible for the child process to execute after termination of the parent in this case.) △

If **_BPX_SHAREAS** is not defined, or has any value other than **YES**, the child process will be executed in a new address space.

If the environment variable **_BPX_SPAWN_SCRIPT** is defined and set to **YES**, the spawn service recognizes an attempt to use **spawn** to invoke a shell script, and instead spawns a copy of the shell to run the script. If the environment variable is not defined, or has some value other than **YES**, the script is treated as a non-executable file.

RETURN VALUE

If successful, **spawn** returns the process id of the new process. If unsuccessful, **spawn** returns **-1**.

USAGE NOTES

The **spawn** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE

```
/* This example must be compiled
   with the posix compiler option */

#include <spawn.h>
#include <unistd.h>
#include <signal.h>

/* posix environment variable ptr */
extern char **environ;
```

```
pid_t subprocess(const char *file,
                 int input_fd,
                 int output_fd,
                 int local)
{
  /*
     This function uses the spawn system
     call to create a subprocess. The
     file descriptor specified by
     input_fd is used as the new process'
     standard input, and the file
     descriptor specified by output_fd
     is used as the new process' standard
     output. If the flag local is non-zero,
     the new process is created in the same
     address space. If local is non-zero,
     the process is created in a new address
     space, and the signal SIGHUP is set to
     be ignored in the new process.
   */

   /* file map for new process */
   int fd_map[3];

   /* inheritance structure */
   struct inheritance inh;

   /* argument vector for new process */
   const char *argv[2];

   /* old handler for SIGHUP */
   void (*hup_hndlr)(int);

   pid_t newpid;

   /* use input_fd as new stdin */
   fd_map[0] = input_fd;

   /* use output_fd as new stdout */
   fd_map[1] = output_fd;

   /* use same file for stderr */
   fd_map[2] = 2;

   inh.flags = SPAWN_SETSIGDEF;

   /* set all signals to default
      in new process */
   sigfillset(inh.sigdefault);

   /* if spawning non-locally */
   if (local == 0)
   {
      /* don't default SIGHUP */
      sigdelset(inh.sigdefault, SIGHUP);
      /* temporarily ignore SIGHUP */
      hup_hndlr = signal(SIGHUP, SIG_IGN);
   }

   setenv("_BPX_SHAREAS",
          local? "YES": "NO");

   /* set up argv for new process, */
   argv[0] = file;

   /* no args */
   argv[1] = 0;

   /* spawn new procdess */
   newpid = spawn(file, 3, fd_map, &inh,
                  argv, environ);

   /* restore SIGHUP handling
      if necessary */
   if (local == 0)

      signal(SIGHUP, hup_hndlr);

   /* return id of new process */
   return newpid;
}
```

RELATED FUNCTIONS

execve, **fork**, **oeattache**, **spawnp**

spawnp

Spawn a new process

SYNOPSIS

```
#include <spawn.h>
pid_t spawnp(const char *file,
             int count,
             const int fd_map[],
             struct inheritance *inh,
             const char *argv[],
             const char *envp[]);
```

DESCRIPTION

The **spawnp** function creates a new process to run an executable program in the hierarchical file system. It performs the same functions as the **spawn** function, except the path portion of the filename is optional.

The **file** argument to **spawnp** is the name of the HFS file to be executed. If the name includes path information, then the file is invoked exactly as specified. If no path information is specified, a search is made of the directories specified by the **PATH** environment variable (as specified by the **envp** argument) until an executable file of the specified name is located.

All other arguments to **spawnp** are interpreted exactly like the corresponding argument to **spawn**.

RETURN VALUE

If successful, **spawnp** returns the process id of the new process. If unsuccessful, **spawnp** returns **-1**.

USAGE NOTES

The **spawnp** function can only be used with MVS 5.2.2 or a later release.

RELATED FUNCTIONS

execve, **fork**, **oeattache**, **spawn**

tcgetsid

Get session leader id for a process

SYNOPSIS

```
#include <sys/types.h>
#include <termios.h>
pid_t tcgetsid(int fileDescriptor);
```

DESCRIPTION

The **tcgetsid** function returns the process group id of the session for which a specific file descriptor is the controlling terminal. The argument **fileDescriptor** specifies the terminal file descriptor for which the information is required.

RETURN VALUE

tcsetgid returns the process group id for the associated session, or **-1** if it fails (for instance, if the file associated with **fileDescriptor** is not a controlling terminal).

USAGE NOTES

The **tcgetsid** function can only be used with MVS 5.2.2 or a later release.

EXAMPLE

This example is compiled using **sascc370 -Krent -o**. This program demonstrates the use of the Open Edition function **tcgetsid()**.

```
/*----------------------------------+
| POSIX/UNIX header files           |
+----------------------------------*/
#include <sys/types.h>

#include <termios.h>

#include <errno.h>

/*----------------------------------+
| ISO/ANSI header files             |
+----------------------------------*/
#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

/*----------------------------------+
| Name:      main                   |
| Returns:   exit(EXIT_SUCCESS) or  |
|            exit(EXIT_FAILURE)     |
+----------------------------------*/
int main()
{
/* session leader id for stdin          */
   pid_t stdin_SID;

/* session leader id for stdout         */
   pid_t stdout_SID;

/* session leader id for stderr         */
   pid_t stderr_SID;

/*--------------------------------------*/
/* Get the Session Leader ID for STDIN    */
/*--------------------------------------*/
printf("\nGet the Session Leader ID
  for STDIN\n");

   stdin_SID = tcgetsid(STDIN_FILENO);

   /* Test for Error */
   if (stdin_SID == -1)
   {
   fprintf(stderr,"Could not get SID
   for stdin\n");
   exit(EXIT_FAILURE);
   }
   else
   {
   printf(" The Session Leader ID
   for stdin: %d\n", (int)stdin_SID);
   }

/*-------------------------------------------*/
/* Get the Session Leader ID for STDOUT       */
*-------------------------------------------*/
printf("\nGet the Session Leader ID for STDOUT\n");

   stdout_SID = tcgetsid(STDOUT_FILENO);

   /* Test for Error */
   if (stdout_SID == -1)
   {
   fprintf(stderr,"Could not get SID
   for stdout\n");
      exit(EXIT_FAILURE);
   }
   else
   {
   printf("The Session Leader ID
   for stdout: %d\n", (int)stdout_SID);
   }

/*-------------------------------------------*/
/* Get the Session Leader ID for STDERR        */
/*-------------------------------------------*/
printf
  ("\nGet the Session Leader ID for STDERR\n");

   stderr_SID = tcgetsid(STDERR_FILENO);

   /* Test for Error */
   if (stderr_SID == -1)
   {
   fprintf(stderr,"Could not get SID
   for stderr\n");
   exit(EXIT_FAILURE);
   }
   else
```

```
    {
    printf("The Session Leader ID
    for stderr: %d\n", (int)stderr_SID);
    }

    exit(EXIT_SUCCESS);

}  /* end of main() */
```

RELATED FUNCTIONS
getsid, **setsid**, **tcgetpgrp**

truncate

Truncate an HFS file

SYNOPSIS

```
#include <unistd.h>
int truncate(const char *pathname,
             off_t length);
```

DESCRIPTION
truncate truncates or extends an OpenEdition file. **pathname** specifies the name of the file. **length** specifies the size of the file. If the length is greater than the current file size, zero bytes are added to the file at the end.

For programs not compiled with the posix option, a style prefix may be required as part of the **pathname**. See "File Naming Conventions" in the *SAS/C Library Reference, Volume 1*, for further information.

RETURN VALUE
truncate returns **0** if successful, or **-1** if unsuccessful.

USAGE NOTES
The **truncate** function can only be used with MVS 5.2.2 or a later release.

RELATED FUNCTIONS
ftruncate

Enhanced OpenEdition Library Functions

This section describes the SAS/C OpenEdition library functions enhanced by Release 6.50.

alarmd

As a result of changes in the MVS operating system, the **alarmd** function description for Release 6.50 requires an update in the CAUTION Section of the "Function Descriptions" in Chapter 6 of the *SAS/C Library Reference, Volume 1, Release 6.00* on page 6–32.

Replace the following sentence under CAUTION:

Old If **SIGALRM** is handled by OpenEdition, **alarmd** is not available.

New If **SIGALRM** is handled by OpenEdition, **alarmd** is not available in releases of MVS prior to MVS 5.2.2.

siglongjmp and blkjmp

The **sigsetjmp** and **siglongjmp** functions, introduced in SAS/C Release 6.00, allow the signal mask to be saved as part of a **setjmp** operation, and restored as part of a **longjmp** operation. In the 6.00 implementation, **siglongjmp** restored the signal mask before searching the stack for **blkjmp** callers.

This meant that the old signal mask was restored before any caller of **blkjmp** received control to intercept the jump. Since in most cases the restored signal mask allows more signals than the old signal mask, this had the effect of allowing a signal to be discovered in a **blkjmp** cleanup routine, thereby causing part of the cleanup to be bypassed.

The 6.50 version of the library modifies **siglongjmp** so that the signal mask is changed as late as possible. If there is no interference from **blkjmp** callers, the signal mask is changed immediately before control is returned to the target **sigsetjmp** call.

As with the 6.00 library, if a call to **siglongjmp** is intercepted by **blkjmp**, the signal mask is restored immediately before control is returned to the **blkjmp** call.

Additionally, a new function called **sigblkjmp** has been defined. This function is an enhanced version of **blkjmp**, which stores the signal mask data associated with a **siglongjmp** in the buffer passed to **sigblkjmp** as well as the registers and other environmental information.

This means that the signal mask is not changed when control is given to a **sigblkjmp** cleanup routine. The mask is only changed when control passes to the original **sigsetjmp** call, or to a caller of the old **blkjmp** function.

Use of **sigblkjmp** rather than **blkjmp** is recommended in any program which uses **sigsetjmp** and **siglongjmp**.

Note: Both **blkjmp** and **sigblkjmp** are compatible with both the **setjmp** and **longjmp** functions as well as with their **sig-** versions. △

sleepd

As a result of changes in the MVS operating system, the **sleepd** function description requires an update in the DESCRIPTION Section of the "Function Descriptions" in Chapter 6 of the *SAS/C Library Reference, Volume 1, Release 6.00* on page 6–434.

Replace the following sentences under DESCRIPTION:

Old If **SIGALRM** is managed by OpenEdition, the **sleep** function is implemented by OpenEdition, and the **sleepd** function is not implemented. In this case, note that the occurrence

of a signal managed by SAS/C does not cause **sleep** to terminate.

New If **SIGALRM** is managed by OpenEdition, the **sleep** and **sleepd** functions are implemented by OpenEdition. In this case, note that the occurrence of a signal managed by SAS/C does not cause **sleep** or **sleepd** to terminate.

tcsetpgrp

IBM has changed the way the OpenEdition **tcsetpgrp** system call works. As a result, several changes for Release 6.50 are required to the existing documentation, *SAS/C Library Reference, Volume 2, Release 6.00*.

First, the **tcsetpgrp** function needs an addition note in the DESCRIPTION Section on page 20–96 to clarify the process handling of the **SIGTTOU** signal.

Replace the following sentence under DESCRIPTION:

Old Note not required.

New *Note:* If **tcsetpgrp** is called from a background process, the signal **SIGTTOU** is generated, unless the signal is ignored or blocked. If the signal is defaulted, this will cause the calling process to stop. If the signal is handled, **tcsetpgrp** will set **errno** to **EINTR** and fail. For this reason, you should ignore or block **SIGTTOU** if you call **tcsetpgrp** from a background process. △

Second, a minor typographical error was discovered in last paragraph of the **tcsetpgrp** DESCRIPTION Section on page 20–96 and was not corrected by the errata sheet for Volume 2.

Make the following correction under DESCRIPTION:

Old SITTIN

New SIGTTIN

Lastly, due to its level of complexity, the example for **setpgid,** which uses **tcsetpgrp,** requires modifications. The **setpgid** example is located in the *SAS/C Library Reference, Volume 2, Release 6.00* in the middle of page 20-75.

Replace the following lines under EXAMPLE:

Old

```
/* become foreground process group */
if (tcsetpgrp(STDIN_FILENO, getpid()))
     erreport("tcsetpgrp");
```

New

```
/* ignore SIGTTOU during tcsetpgrp */
signal(SIGTTOU, SIG_IGN);
/* become foreground process group */
if (tcsetpgrp(STDIN_FILENO, getpid()))
      erreport("tcsetpgrp");
/* restore normal SIGTTOU handling */
signal(SIGTTOU, SIG_DFL);
```

New Signals for OpenEdition

Release 6.50 of SAS/C has added support for a number of new signals. These signals are defined by OpenEdition MVS in Release 5.2.2 and later. Because these signals are implemented using OpenEdition, it is necessary for programs not called by the MVS shell to call the **oesigsetup** function in order to handle them.

Many of these signals (such as **SIGBUS**) are defined for compatibility with UNIX operating systems, but do not have a defined meaning under MVS. For instance, under some versions of UNIX, a program would catch the **SIGBUS** signal to handle memory access errors that cause a **SIGSEGV** exception under MVS. Consult UNIX documentation for further information on these signals.

The new signals and their meanings are as follows:

SIGBUS – UNIX compatibility
SIGIOER – IBM C compatibility
SIGPOLL – UNIX compatibility
SIGPROF – profiling timer expired (see **setitimer**)
SIGSYS – UNIX compatibility
SIGURG – UNIX compatibility
SIGVTALRM – virtual timer expired (see **setitimer**)
SIGWINCH – UNIX compatibility
SIGXCPU – CPU time limit exceeded (see **setrlimit**)
SIGXFSZ – file size limit exceeded (see **setrlimit**)

The default actions for **SIGIOER** and **SIGWINCH** are to ignore the signal. For all others, the default is process termination.

CHAPTER 3

SAS/C Library Changes in Release 6.50

Introduction

This chapter describes the changes and enhancements to additional (Non-OpenEdition) SAS/C Library Functions for Release 6.50.

Release 6.50 Enhancements

The following enhancements to the SAS/C Library have been implemented with Release 6.50:

- **DOM** – Delete Operator Message
- **DOM_TOK** – Delete Operator Message with Tokens
- **WTO** – Write to Operator
- **WTOR** – Write to Operator with Reply
- **addsrch** – New MVS Search Capability
- new amparms:

  share=ispf

  share=alloc

  share=rls

  share=rlsread

These enhancements are supported by the Systems Programming Environment (SPE) and the full SAS/C library.

New (Non-OpenEdition) Library Functions

This section describes the changes and enhancements to additional (Non-OpenEdition) SAS/C Library Functions for Release 6.50.

DOM

Delete Operator Message

SYNOPSIS

```
#include <oswto.h>
void DOM(int iMsg);
```

DESCRIPTION

The **DOM** function implements the functionality of the MVS assembler **DOM** macro. This function is used to delete an operator message from the display screen of the operator's console. It can also prevent messages from ever appearing on any operator's console. When a program no longer requires that a message be displayed, it can issue the **DOM** function to delete the message. The **iMsg** argument is returned as a fullword from the **WTO** or **WTOR** function, which has been coded with the **_Wmsgid** keyword.

RETURN VALUE

There is no return value from the **DOM** function.

IMPLEMENTATION

The **DOM** function is implemented by the source module **L$UWTO**.

EXAMPLE

This example uses the **DOM** function to delete an operator message.

```
#include <oswto.h>

int iMsg;

WTO("Experiencing storage shortage",
    _Wmsgid &iMsg, _Wend);
          .
          .
          .
/* delete msg from console */
```

```
DOM(iMsg);
```

RELATED FUNCTIONS

DOM_TOK, **WTO**, **WTOR**

DOM_TOK

Delete Operator Message (using token)

SYNOPSIS

```
#include <oswto.h>
void DOM_TOK(int iMsg);
```

DESCRIPTION

The **DOM_TOK** function implements the functionality of the MVS assembler **DOM** macro. This function is used to delete an operator message from the display screen of the operator's console. It can also prevent messages from ever appearing on any operator's console. When a program no longer requires that a message be displayed, it can issue the **DOM_TOK** function to delete the message. The **iMsg** argument is same as the fullword supplied to the **WTO** or **WTOR** functions using the **_Wtoken** keyword.

RETURN VALUE

There is no return value from the **DOM_TOK** function.

IMPLEMENTATION

The **DOM_TOK** function is implemented by the source module **L$UWTO**.

EXAMPLE

This example uses the **DOM_TOK** to delete an operator message.

```
#include <oswto.h>

int iMsg;

WTO("Experiencing storage shortage",
    _Wtoken 12345, _Wend);
          .
          .
          .
/* delete msg from console */
DOM_TOK(12345);
```

RELATED FUNCTIONS

DOM, **WTO**, **WTOR**

WTO

Write to Operator

SYNOPSIS

```
#include <oswto.h>
int WTO(char *msg, ...);
```

DESCRIPTION

The **WTO** function implements the functionality of the MVS assembler **WTO** macro. The **msg** argument is the address of a null-terminated string, or in the case of a multi-line message, this argument should be set to **0**. The remainder of the argument list is a list of keywords followed, in most cases, by an argument specifying a value for the keyword. The list is terminated by the **_Wend** keyword. The supported keywords and their associated data are as follows:

- The **_Wctext** keyword is equivalent to the Assembler **TEXT=(msg,C)**, which identifies the first line of a multi-line message as a *control* line. The next argument should be a null-terminated string containing the text to be displayed. If coded, this must be the first text type of keyword and can only be specified once. For this and the following text arguments, the first argument to **WTO**, **msg**, must be **0** to indicate a multi-line rather than a single-line message.
- The **_Wltext** keyword is equivalent to the Assembler **TEXT=(msg,L)**, which identifies a *label* line of a multi-line message. The next argument should be a null-terminated string containing the text to be displayed. This argument, if coded, must follow the **_Wctext** argument and precede any **_Wtext** arguments. There can be a maximum of 2 label lines.
- The **_Wtext** keyword is equivalent to the Assembler **TEXT=(msg,D)**, which identifies a *detail* line of a multi-line message. The next argument should be a null-terminated string containing the text to be displayed. Up to 10 detail lines can be output, however if control or label lines are also to be sent, the total number of lines is still limited to 10.
- The **_Wroutcde** keyword is equivalent to the Assembler **ROUTCDE** keyword. The next argument(s) should consist of one or more integers representing routing codes in the range of 1-28.
- The **_Wdesc** keyword is equivalent to the Assembler **DESC** keyword. The next argument(s) should consist of one or more integers representing descriptor codes in the range of 1-13.
- The **_Wresp** keyword is equivalent to the Assembler **MCSFLAG(RESP)** keyword, indicating that the **WTO** is an immediate command response.
- The **_Wreply** keyword is equivalent to the Assembler **MCSFLAG(REPLY)** keyword, indicating that this **WTO** is a reply to a **WTOR**.
- The **_Wbrdcst** keyword is equivalent to the Assembler **MCSFLAG(BRDCST)** keyword, used to broadcast the message to all active consoles.
- The **_Whrdcpy** keyword is equivalent to the Assembler **MCSFLAG(BHRDCPY)** keyword, indicating to queue the message for hard copy only.
- The **_Wnotime** keyword is equivalent to the Assembler **MCSFLAG(NOTIM)** keyword, indicating that the time should not be appended to this message.
- The **_Wcmd** keyword is equivalent to the Assembler **MCSFLAG(CMD)** keyword, indicating that the **WTO** is a

recording of a system command issued for hardcopy log purposes.

- The **_Wbusyexit** keyword is equivalent to the Assembler **MCSFLAG(BUSYEXIT)** keyword, indicating that if there are no message or console buffers, the **WTO** is not to go into a wait state.
- The **_Wcart** keyword is equivalent to the Assembler **CART** keyword. The next argument should be the address of an 8-byte field containing a command and response token to be associated with this message.
- The **_Wkey** keyword is equivalent to the Assembler **KEY** keyword. The next argument should be the address of an 8-byte key to be associated with this message.
- The **_Wtoken** keyword is equivalent to the Assembler **TOKEN** keyword. The next argument should be an unsigned long integer representing the token to be associated with this message. This is used to identify a group of messages that can be deleted by the **DOM_TOK()** function. The token must be unique within an address space.
- The **_Wconsid** keyword is equivalent to the Assembler **CONSID** keyword. The next argument should be an unsigned long integer containing the id of the console to receive the message. This argument is mutually exclusive with **_Wconsname**.
- The **_Wconsname** keyword is equivalent to the Assembler **CONSNAME** keyword. The next argument should be the address of an 8-byte field containing a 2 through 8 character name, left-justified and padded with blanks naming the console to receive the message. This argument is mutually exclusive with **_Wconsid**.
- The **_Wmsgid** keyword is to pass back a message identification number if the **WTO** is successful. The next argument should be the address of an unsigned long variable which will be filled in after the **WTO** completes. This number can then be used to delete the **WTO** with the **DOM()** function.
- The **_Wend** keyword indicates the end of the list of keywords.

RETURN VALUE

WTO returns **0** if the **WTO** macro was successful. If the **WTO** macro fails, it returns the return code from the macro, which will be a positive value. **WTO** may also return **-1** to indicate an unknown or invalid keyword combination, or **-2** if there was not enough memory to perform the **WTO**.

IMPLEMENTATION

The **WTO** function is implemented by the source module **L$UWTO**. As a convenience, the macro **WTP** can be used for single line messages used by programmers. It is defined as follows:

```
#define WTP(msg) WTO(msg,
                     _Wroutcde, 11,
                     _Wdesc, 7,
                     _Wend);
```

EXAMPLES

EXAMPLE 1:

This example uses the **WTP** macro to send two single-line programmer's messages:

```
#include <oswto.h>

char msg[120];
int iLine;

iLine = 20;
sprintf(msg, "Error discovered at line:
        %i", iLine);
WTP(msg);
WTP("Aborting...");
```

EXAMPLE 2:

This example sends a multi-line message to a specific console.

```
#include <oswto.h>
#include <code.h>
#include <genl370.h>

int regs[16];
char line1[50];
char line2[50];
char line3[50];
char line4[50];

_ldregs(R1, regs);

/* save current registers
   in the regs array */
STM(0,15,0+b(1));

sprintf(line1,
   "GPR  0-3  %08X  %08X  %08X  %08X",
   regs[0], regs[1], regs[2], regs[3]);

sprintf(line2,
   "GPR  4-7  %08X  %08X  %08X  %08X",
   regs[4], regs[5], regs[6], regs[7]);

sprintf(line3,
   "GPR  8-11 %08X  %08X  %08X  %08X",
   regs[8], regs[9], regs[10], regs[11]);

sprintf(line4,
   "GPR 12-15 %08X  %08X  %08X  %08X",
   regs[12], regs[13], regs[14], regs[15]);

WTO(0, _Wctext, "XXX99999",
       _Wtext, "Register contents:",
       _Wtext, line1,
       _Wtext, line2,
       _Wtext, line3,
       _Wtext, line4,
       _Wconsname, "CONSOLE1",
       _Wroutcde, 11,
```

```
        _Wdesc, 7,
        _Wend);
```

RELATED FUNCTIONS
DOM, **DOM_TOK**, **WTOR**

WTOR

Write to Operator with Reply

SYNOPSIS

```
#include <oswto.h>
int WTOR(char *msg, ...);
```

DESCRIPTION
The **WTOR** function implements the functionality of the MVS assembler **WTOR** macro. The **msg** argument is the address of a null-terminated string. The remainder of the argument list is a list of keywords followed, in most cases, by an argument specifying a value for the keyword. The list is terminated by the **_Wend** keyword. The supported keywords and their associated data are as listed under the **WTO** function with the addition of the following:

- The **_Wreplyadr** keyword is to pass back the operator reply. The next argument should be the address of an area which will be filled in after the **WTOR** completes. This area can be from 1 to 119 bytes in length.
- The **_Wreplylen** keyword is used to indicate the length of the **_Wreplyadr** area. The next argument should be an unsigned character variable containing an integer in the range of 1 to 119.
- The **_Wecb** keyword is used to identify an ECB to be **POST**ed when the **WTOR** has received a response. The next argument should be a pointer to a fullword, which is the ECB to be **POST**ed.
- The **_Wreplycon** keyword is equivalent to the Assembler **RPLYISUR** keyword. The next argument should be the address of a 12-byte field where the system will place the 8-byte console name and the 4-byte console id of the console through which the operator replies to this message.

RETURN VALUE
WTOR returns **0** if the **WTOR** macro was successful. If the **WTOR** macro fails, it returns the return code from the macro, which will be a positive value. **WTOR** may also return **-1** to indicate an unknown or invalid keyword combination, or **-2** if there was not enough memory to perform the **WTOR**.

IMPLEMENTATION
The **WTOR** function is implemented by the source module **L$UWTO**.

EXAMPLE
This example uses the **WTOR** to request some information from the operator. The program then waits for a response and then writes the response back to the operator console.

```
#include <oswto.h>
#include <ostask.h>

int msgid;
unsigned int uiEcb;
unsigned char ucConsoleName[13]
                        = "            ";
unsigned char ucOperatorReply[71];
memset(ucOperatorReply, ' ', 70);
WTOR("Enter password:",
     _Wreplyadr, ucOperatorReply,
     _Wreplylen, 70,
     _Wecb, &uiEcb,
     _Wreplycon, ucConsoleName,
     _Wtoken, 12345,
     _Wend);
ucOperatorReply[70] = 0;
ucConsoleName[8] = 0;
WAIT1(&uiEcb);
WTO(0, _Wctext, "XXX99999",
    _Wtext, "Operator at console:",
    _Wtext, ucConsoleName,
    _Wtext, "replied:",
    _Wtext, ucOperatorReply,
    _Wroutcde, 11,
    _Wdesc, 7,
    _Wend);
```

RELATED FUNCTIONS
DOM, **DOM_TOK**, **WTO**

New MVS Search Capability

addsrch

Indicate a location from which modules may be loaded

SYNOPSIS

```
#include <dynam.h>

SEARCH_P addsrch(int type,
                 const char *loc,
                 const char *prefix);
```

DESCRIPTION
addsrch adds a location to the list of locations from which modules can be loaded. This list controls the search order for modules loaded from a call to **loadm**. **addsrch** does not verify the existence of the location.

The first argument, **type**, must be a module type defined in **<dynam.h>**. The module type defines what type of module is loaded and can vary from operating system to operating system. The character string specified by the second argument, **loc**, names the location. All location strings may have leading and trailing blanks, and all characters are converted to an

uppercase format. The format of this string depends on the module type.

The search order can be described additionally by the third argument, **prefix**. The **prefix** argument is a character string of no more than eight characters. **prefix** may be null (""), but if it not, then it specifies that the location indicated is searched only if the load module name (as specified by the **type** argument to **loadm**) begins with the same character or characters specified in **prefix**.

Under MVS, the module type, **type**, controls the format of the second argument, **loc**, which names the location to be searched by **loadm**. The module type may be either **MVS_DD** or **MVS_DSN**:

- **MVS_DD** – The location parameter must be a previously allocated DDname, either through JCL or dynamically.
- **MVS_DSN** – The location parameter must be a fully qualified dataset name.

Under CMS, the defined module types for the first argument, **type**, are the following:

- **CMS_NUCX** – Specifies that the module is a nucleus extension. The module has been loaded (for example, by the CMS command NUCXLOAD) before **loadm** is called.
- **CMS_LDLB** – Specifies that the module is a member of a CMS LOADLIB file. The LOADLIB file must be on an accessible disk when **loadm** is called.
- **CMS_DCSS** – Specifies that the module resides in a named segment that has been created using the GENCSEG utility, as documented in "The CMS GENCSEG Utility" in Appendix 3, of the *SAS/C Compiler and Library User's Guide*.

The module type also controls the format of the second argument, **loc**. The **loc** argument identifies the location to be searched by **loadm**. For the following the module types, the **loc** argument is:

CMS_NUCX
The location parameter must be null ("").

CMS_LDLB
The location parameter is the filename and filemode of the **LOADLIB** file in the form *filename filemode*, for example, DYNAMC A1 specifies the file, DYNAMC LOADLIB A1. The *filemode* may be an asterisk (*).

CMS_DCSS
The location parameter is a 1–8 character string that names the segment. An asterisk (*) as the first character in the name is used to specify that the segment name is for a non-shared segment.

At the C program's initialization, a default location is in effect. The default location is defined by the following call:

```
sp = addsrch(CMS_LDLB, "DYNAMC *","")
```

RETURN VALUE
addsrch returns a value that can be passed to **delsrch** to delete the input source. This is a value of the defined type **SEARCH_P**, which can be passed to **delsrch** to remove the location from the search order. If an error occurs, a value of 0 is returned.

USAGE NOTES
addsrch does not verify that a location exists or that load modules may be loaded from that location. The **loadm** function searches in the location only if the load module cannot be loaded from a location higher in the search order. **addsrch** fails only if its parameters are ill-formed.

EXAMPLE

```
#include <dynam.h>

SEARCH_P mylib;
      .
      .
      .
/* Search for modules in a CMS LOADLIB. */
mylib=addsrch(CMS_LDLB, "PRIVATE *", "");

/* Search for modules in a MVS dataset. */
mylib=addsrch(MVS_DSN, "SYS1.LINKLIB", "");
```

New Access Method Parameters (amparms)

This section describes the new file usage amparms for Release 6.50.

The share=ispf amparm

The **share=ispf** amparm allows a program to write to an ISPF member without allocating the entire dataset as "OLD". Other programs can continue to read and write to other members while the program updates the designated member.

Here is an example:

```
int cardfd;
cardfd = aopen("dsn:sas.test.c(hello)",
               O_RDWR, "share=ispf");
```

Note: Opening a file with the **share=ispf** amparm allows a PDS to be shared by several programs or users but must be used carefully.

Using **share=ispf** allows a PDS to be allocated as SHR and used by cooperating programs, that is, by the ISPF editor and utilities, by other SAS/C programs which open specifying **share=ispf**, and by any other applications which observe the ISPF protocols.

Using **share=ispf** does not prevent access by applications that do not observe the ISPF protocols. Such access may cause file damage or loss of data.

While a SAS/C program has a PDS member open with **share=ispf**, an attempt by an ISPF user to save another member of the same PDS will wait until the SAS/C program closes the member. Similarly, when

one SAS/C program has a PDS member open using **share=ispf**, any other SAS/C program which opens the same PDS with **share=ispf** will wait until the first program closes its member. For this reason, programs which use **share=ispf** should be designed to keep such files open for as small an interval of time as possible. △

The share=alloc amparm

The **share=alloc** amparm is used to open a new or existing file by DSN as shared. Here is an example:

```
int cardfd;
cardfd =
  aopen("dsn:sas.test",
        O_CREAT | O_RDWR | O_APPEND,
       "recfm=f,reclen=80,share=alloc");
```

CAUTION:

When you open a file using the share=alloc amparm, the operating system and the C library offer little protection against file damage or loss of data if several programs write to the file at the same time. Some versions of MVS will ABEND a program which attempts to write to a PDS that another program has opened; however, no protection at all is available for sequential data sets. For this reason, the **share=alloc** amparms should be used only when there is no risk of multiple simultaneous access, when the program itself synchronizes access to the file (for example, using the MVS ENQ macro, or where the risk of occassional loss of data or file damage is considered acceptable. △

Record Level Sharing (rls) amparms

shr=rls

specifies the use of VSAM Record Level Sharing protocols for processing the dataset.

shr=rlsread

specifies the use of VSAM Record Level Sharing protocols for processing the dataset and specifies their use for nonupdate file accesses.

Note: Many of the sharing pitfalls can be prevented by used of VSAM Record Level Sharing (RLS) support available for MVS with DFSMS Version 1, Release 3 or higher. Also, with VSAM, RLS locking is done at the record level as opposed to the control interval. △

The **share=rls** or **share=rlsread** amparm specifies that VSAM record level sharing protocols are to be used for processing the dataset. The RLS protocols are available for MVS only with DF/SMS Version 1, Release 3 (or higher) installed or OS/390 Release 2, which includes it. In addition, using the RLS feature requires supporting hardware (SYSPLEX Coupling Facility (CF)) and proper site installation configuration of DF/SMS.

This option is meaningful only for VSAM files and is equivalent to coding **MACRF=(RLS)** for **share=rls** or **MACRF=(RLS)**, **RLSREAD=CRI** for **share=rlsread** on an ACB assembler macro used to open the VSAM file. RLS implies the use of cross-system record level locking as opposed to CI locking, uses CF for cross-system buffer consistency with a coordinated system wide local buffer cache. The amparm **share=rlsread** also specifies that consistent read integrity (record locking during all reads) be used for read requests, even if the dataset is opened for read only (**mode=r**) or **K_noupdate** flag is specified with **kretrv**. Specifying **share=rlsread** overrides any RLS JCL specification. VSAM RLS supports only cluster or path level access (that is, no individual data, index, or alternate index component access) and does not support linear datasets or datasets which are defined with an imbedded index or a keyrange. Recoverable datasets, that is, datasets defined with the IDCAMS LOG(UNDO) or LOG(ALL) attribute, can only be opened for read (**mode=r**) because of the CICS file control and logging requirements. Nonrecoverable datasets, that is, LOG(NONE), or the default can be opened in any mode with the LOG(NONE) attribute since no CICS file control or logging is required for these datasets.

share=rls or **share=rlsread** causes VSAM to ignore any bufnd/bufni/bufsp specification. However, in general, the VSAM RLS feature is transparent from a C library VSAM file function acces usage standpoint since the protocols are implemented at the operating system level and not in the library except for specifying the ACB options. However, a couple of pitfalls are worth mentioning. File opens with VSAM RLS will fail if there are nonRLS users of the dataset, and conversely nonRLS opens will fail if there are RLS users of the dataset. While MVS recovery is usually satisfactory, it is possible through system crashes and/or abends that it leaves the dataset locked out from nonRLS access, and locked records unavailable to RLS users until the IDCAMS utility is called to fix it.

CHAPTER

4

SAS/C COOL Changes in Release 6.50

Introduction

This chapter provides a complete description of the changes and enhancements made to the SAS/C COOL pre-linker for Release 6.50.

Release 6.50 Enhancements

The following enhancements to the SAS/C COOL pre-linker have been implemented with Release 6.50:

- Inclusion of debugging information in the object deck with new option:

  dbglib

- Template support – processing of a new object format when automatically instantiated templates are specified
- Marking and detecting previously processed COOL objects with two new options:

  allowrecool

  ignorerecool

- Enhanced option:

  enxref

Table 4.1 on page 63 lists the new options available for the COOL pre-linker and the systems to which these options apply. Descriptions of each option can be found in the following sections.

Table 4.1 New COOL Options

Option	TSO	CMS	MVS Batch	OpenEdition
allowrecool	X	X	X	X
dbglib	X	X	X	X
ignorerecool	X	X	X	X

Inclusion of Debugging Information in the Object Deck

In Release 6.50, the compiler allows for 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 **autoinst** 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.

dbglib Option

The **dbglib** option specifies a debugger file qualifier that provides for customization of the destination of the debugger file.

For each platform, **dbglib** specifies something different:

ON MVS:
A SAS/C file specification that denotes a PDS. The filename is constructed using whatever is supplied, followed by (***sname***).

On CMS:
If the option specified starts with a ' **/**', then it is assumed that this is either a ' **//sf:**' file specification or an SFS path. In this case, the specification is prepended to the filename. For example,

```
dbglib(//sf:ted/)
```

will generate the name

```
//sf:/ted/sname.DB
```

If the option specified does not start with a '**/**' then it is considered to be a filemode, and will be appended to the filename. For example,

```
dbglib(d2)
```

will generate the name

```
sname.db.d2
```

On OE:
The option specified is a path name to be prepended to the filename. For example,

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

will generate a filename of

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

The option has different defaults on the various platforms:

ON MVS:
dbglib(ddn:sysdblib)

On CMS:
dbglib(A)

On OE:
dbglib()

For the various platforms, the default filename has different forms:

On MVS:
ddn:sysdblib(*sname*)

On CMS:
***sname*.DB.A**

On OE:
hfs:*sname*.dbg370

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

The short form of this option is **-db**.

Template Support

In Release 6.50, the compiler allows for the generation of automatically instantiated template functions, when the **autoinst** compiler option is specified. When this option is specified, the compiler uses a "shelled object" format containing the output of the primary compilation and all template functions needed by that compilation. In this release, COOL has been modified to process this new object format and the "shelled" template functions.

When a "shelled object" is encountered by COOL, the primary object deck is processed, and any template function objects are processed if a template function by the same name has not already been processed. This results in the inclusion of the first template function found with a given name.

Note: "shelled objects" are specified in the same manner as any other object deck. △

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, COOL produces a diagnostic message indicating the condition.

The new processing is divided into two phases. The first phase marks the output object deck to indicate it has already been processed with COOL. It is controlled by the **allowrecool** and **noallowrecool** options. The second phase detects that an input object deck has been marked to indicate it was previously processed. The second phase is controlled by the **ignorerecool** and **noignorerecool** options. By default, COOL marks the object deck to prevent an attempt to reprocess it. Also by default, COOL detects that the input object deck was previously processed by COOL.

Note: These defaults can cause COOL to indicate an error where it would not detect such an error in previous releases. Under certain restricted circumstances, it is possible to generate object code that can be successfully processed by COOL more than once. If this behavior is desired, the options can be specified such that the output object's decks are not marked and/or that such marking be ignored. △

allowrecool Option

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.

The short form for this option is **-rc**.

Note: COOL does not modify the object deck to enable reprocessing. It is the user's responsibility to determine if a particular object is eligible for reprocessing. △

ignorerecool Option

The **ignorerecool** option specifies that if any marks are detected indicating that COOL has already processed an input object deck, then the marks are to be ignored. If the **ignorerecool** option is specified along with the **verbose** option, then a diagnostic message is issued and processing continues.

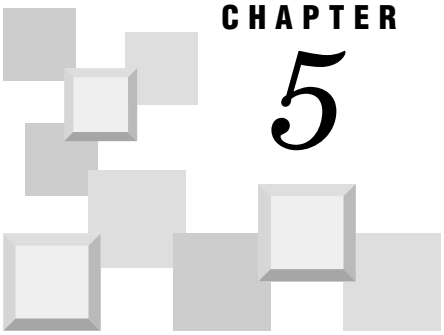
The default **noignorerecool** specifies that any mark indicating that COOL has already processed an input object deck should result in an error message and process termination.

The short form for this option is **-ri**.

The Enhanced enxref Option

enxref(-Areferences under OpenEdition)

For Release 6.50, when the **references** option is specified for **enxref**, referenced symbols as well as defined symbols are included in the cross-reference listing.



CHAPTER 5

SAS/C Debugger Changes in Release 6.50

Introduction

This chapter provides a complete description of the changes and enhancements made to the SAS/C Debugger for Release 6.50.

Release 6.50 Enhancements

In Release 6.50, a change has been made to re-attempt the **set search** logic for a particular function when a previous search for the files associated with this functin had failed. Prior to Release 6.50, only one attempt would be made to read the debugger or source files.

The following window customization command has been added for Release 6.50:

sqbracket – square bracket

Incompatibility with Previous Releases

Release 6.50 of the SAS/C Debugger is not compatible with debugging files generated by the SAS/C Compiler prior to Release 5.50.

Search Lists

This release 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, that behavior has been modified, such that, if a **set search** is issued, followed by a **list** command, then the degugger will attempt 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 failed because the source files had been moved to the dataset SASC.APPL.SOURCE, then issuing the command :

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

followed by a **list** command would cause the debugger to reattempt the search for the source.

Note that 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.

Configuration File Window Customization

This section describes the changes and enhancements for the 3270 square bracket customization in the Config window in Release 6.50.

sqbracket

The new **sqbracket** subcommand can be used for window customization and is valid in the configuration file. The **sqbracket** command specifies the translation of square bracket characters **[** and **]** to and from their 3270 datastream representation. Both outbound characters can be customized.

3270 Datastream Square Bracket Representation

The 3270 datastream square bracket representation in hexadecimal is:

```
Outbound display [ ] ---> 41 42

Inbound input [ ] <--- 41 42
```

Changing Square Bracket Characters

The 3270 datastream representation of square brackets for both outbound terminal display and inbound terminal input can be customized. For most 3270 debugger users the debugger automatically selects the correct translation based upon the 3270 device information obtained during window initialization. However, in some circumstances, mainly with some 3270 emulators, the translation selected by the debugger may not properly display or input square bracket characters. The square bracket customization, in the Config window view, contains four fields for the left and right square bracket character translations for both outbound terminal display and inbound terminal input. These fields are initially shown with the settings selected by the debugger or configuration file settings and may be modified by entering a two digit hexadecimal value in the range 40–FE as the replacement character. See Table 5.1 on page 68 for possible translation values.

Table 5.1 Display — Translation Table

Display	Translation Values (hex)
3277	none
3278	41 42
3279	ba bb
for certain 3270 emulators	ad bd

CHAPTER

6

SAS/C Debugger Changes in Release 6.00

Introduction

This chapter provides a complete description of the changes and enhancements made to the SAS/C Debugger for Release 6.00.

Release 6.00 Enhancements

The following enhancements to the SAS/C Debugger have been implemented with Release 6.00:

- remote debugging of applications running in a different address space or system, or using a different terminal, including CICS and OpenEdition programs
- a CICS front-end transaction for setting up the remote debugging environment for CICS applications
- the **rsystem** command, for use with the remote debugger, for executing a system command in the environment of the program being debugged
- search lists for identifying and locating files used by the debugger
- a run command file for executing debugger commands at start-up in environments where the debugger does not support REXX.

In addition to these enhancements, there are several minor changes in this release of the SAS/C Debugger, as described in the section "Minor Changes and Enhancements" on page 82.

Incompatibility with Previous Releases

The debugging information produced by Release 6.00 of the SAS/C Compiler has been redesigned. Because of this change, Release 6.00 of the SAS/C Debugger is not compatible with debugging files generated by any releases of the SAS/C Compiler prior to Release 5.50.

Remote Debugger

This release of the SAS/C Debugger includes a new interface for debugging remote applications. The new debugger interface is called the *remote debugger*. The remote debugger allows you to run the debugger display in one process and the program being debugged in another. The processes can be run on the same system or different systems.

Under TSO and CMS, the remote debugger takes the form of a REXX EXEC named **SASCDBG**. Under the MVS OpenEdition shell, it takes the form of an executable named **SASCDBG**.

The remote debugger is intended for use in the following situations:

- when you want to debug a CICS application
- when you want to debug an OpenEdition application
- when you want to run the debugger and application on different terminals or systems.

We recommend using the local debugger for most other debugging tasks, that is, the debugger interface described in *SAS/C Debugger User's Guide and Reference, Third Edition*. For example, when debugging a non-OpenEdition application under TSO, the local debugger is considerably more efficient, since it is not subject to context switching or communication delays.

How the Remote Debugger Works

The remote debugger is similar to the local debugger in its look and feel and operation. The main difference is in the program architecture and start-up procedure.

When running the remote debugger under TSO or CMS, the remote debugger provides the same full-screen debugging capabilities as the local debugger. When running the remote debugger in the OpenEdition shell or under MVS batch, the remote debugger is limited to line-mode operation only.

If you are familiar with the local debugger, as described in *SAS/C Debugger User's Guide and Reference, Third Edition*, you should have no problem using the remote debugger, once you understand the architecture and start-up procedures described here.

Architecture

With Release 6.00, the SAS/C Debugger has a new client/server architecture that allows debugging of remote applications. There are two processes in a remote debugging session:

- the debugger display and program control logic
- the program being debugged and an interface to the debugger (ITD)

The debugger display and program control logic acts as the server. It provides the debugger services, under user control, to the client program. Each debugger component runs in a separate process. Depending on your operating environment, this may mean a different address space, task, virtual machine, or OpenEdition process. In addition, the client program may be running on a physically different host.

As shown in Figure 6.1 on page 70, the debugger processes communicate with each other through a communications layer, which can utilize either the TCP/IP or APPC communications access method. Details on configuring APPC for use with the remote debugger are provided in Appendix 1, "APPC Setup for the Remote Debugger".

For maximum productivity, the debugger display component typically is run in a full-screen session under TSO or CMS. You can run the client program in any environment where the SAS/C library is supported. CICS and OpenEdition processes are the most typical environments. You can also run the client program in environments such as TSO, CMS, MVS APPC address spaces, or MVS batch.

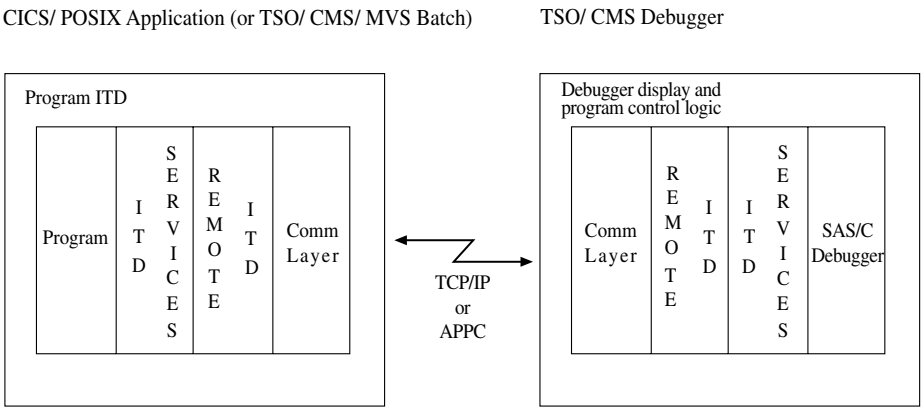
Start-up methods

From the user's point of view, the most significant difference between the local debugger and remote debugger is the start-up procedure. To use the local debugger, you simply compile your program in debug mode and include the **`=debug`** run–time option when you run the executable. (For details, see *SAS/C Debugger User's Guide and Reference, Third Edition*.) The remote debugger has two start-up methods:

- independent start-up
- automatic start-up

With the independent start-up method, you start the debugger display process first, using the **`SASCDBG`** debugger interface. When the process starts, it displays information at the terminal indicating the communications access method and other information needed by the client program to make the connection to the debugger display.

Figure 6.1 Remote debugger architecture



For example, under TSO, you might see the following message after starting the remote debugger display:

```
SASCDBG DBCOMM(TCPIP)

SAS/C Remote Debugger can be reached via:
_DB_COMM=TCPIP connect to _DB_HOST=10.1.1.1
  (MVS), at _DB_PORT=13227
```

Then, in a separate step, you start the program to be debugged in the normal way for your environment, specifying the **=debug** runtime option on the command line. You can specify the connection information with environment variables before starting the program, or with the equivalent command line options. For example, you might call a TSO load module like this:

```
CALL ABC.LOAD(MYPGM) '=D =_DB_COMM=TCPIP
  =_DB_HOST=MVS =_DB_PORT=13227'
```

With the automatic start-up method, you start both processes in a single step by including the program name in the **SASCDBG** command line. For example, in the OpenEdition shell, you might start the remote debugger with this command:

```
sascdbg -tcpip mypgm
```

Both methods rely on environment variables, or the equivalent command line options, to determine the communications access method and other information needed to establish communication between the debugger processes. The section "Debugger Environment Variables" on page 71 describes these variables. The section "Using the SASCDBG Debugger Interface" on page 72 provides a complete description of **SASCDBG** syntax.

CICS applications are a special case. Since CICS does not support a command line, you must use the remote debugger's CICS front-end transaction to set the environment variables and launch the program to be debugged. The section "Debugging CICS Applications" on page 75 describes this procedure.

Other start-up methods

For applications with unique start-up requirements or where automatic start-up is unsupported (for example, MVS batch), you can supply your own *program invocation exit* for starting the program to be debugged. A program invocation exit can be written in C or assembly language and must conform with the specifications described in Appendix 2, "Remote Debugger User Exits".

Debugger Environment Variables

With this release, the SAS/C Debugger inspects one or more environment variables to determine the debugger operating mode: local or remote. In a remote debugging session, both debugger processes inspect these variables to determine the communications access method and the information needed to establish communication between the debugger processes. The following sections describe these environment variables and how to set them.

Environment variable descriptions

_DB_COMM=TCPIP | TCPIP_*xxx* | APPC | NONE | LOCAL

Determines the debugger operating mode – local or remote – and the communications access method for remote operation. Specifying TCPIP, TCPIP_*xxx*, or APPC selects remote operation and specifies whether to use TCP/IP or APPC as the communication method between the debugger processes. When TCPIP_*xxx* is specified, it selects the TCP/IP implementation specified by the **setsockimp("***xxx* **")** function call. The default value is NONE, which starts a local debugging session. LOCAL and NONE are synonyms.

_DB_HOST=*ip_addr* | *hostname*

(TCP/IP only) Dotted decimal IP address or host name of the machine where the remote debugger display process will be run or is running. During independent start-up, this value is displayed at the terminal and is used by the client program to connect to the debugger display.

Note: The host name is actually translated to an IP address by a **gethostbyname** function call. △

_DB_PORT=*port_number*

(TCP/IP only) TCP/IP port number of the debugger display and control process. During independent start-up, this value is displayed at the terminal and is used by the client program to connect to the debugger display.

_DB_LU=*lu_name*

(APPC only) SNA Network Logical Unit (LU) name where the debugger display process will be run or is running. The LU name consists of 1-8 non-blank uppercase letters or numbers (A-Z, 0-9). During independent start-up, the LU name is displayed at the terminal and is used by the client program to request an APPC connection to the debugger display. The default value is 8 blanks, which means to use the system base LU.

Note: The default value may or may not work, depending on the APPC definitions for your system. △

_DB_TP=*tp_name*

(APPC only) Specifies the Transaction Program (TP) name of the debugger display process. This name is registered with the APPC LU, and consists of 1-64 uppercase or lowercase alphabetic characters, numbers, or special characters, except $, #, and @. During independent start-up, the TP name is displayed at the terminal and is used by the client program to request an APPC connection to the debugger display. The default value is **SASCDBG**.

_DB_MODE=*mode_table* | ISTINCLM

(APPC only) Specifies the APPC logon mode table. This table establishes the VTAM session parameters for the LU partners, and need only be specified when debugging a CICS application or when running the client program on a physically different system. You can obtain the mode table specification from the VTAM APPL definition for the LU. You can also use the mode

table ISTINCLM, which is a default system-supplied mode table for VTAM. You must specify the ISTINCLM table explicitly; **_DB_MODE** does not default to this value.

_DB_TIMEOUT=*n*
Timeout value, in seconds, for interprocess communications. A process will be considered non-responsive if it fails to acknowledge a request within the specified period. A message is displayed at the terminal if a communication request times out.

Setting environment variables

You can set the debugger environment variables at three different levels, depending on the system:

- CMS and TSO support environment variables with permanent, external, and program scopes.
- CICS supports external and program scopes.
- MVS batch supports program scope only.

There is no equivalent to variable scopes in the OpenEdition shell. However, OpenEdition environment variables are similar to external scope variables, since they remain defined for the duration of the shell and can be inherited from the shell. For details on environment variable scopes, refer to Chapter 12, "System Interface and Environment Variables," in *SAS/C Library Reference Third Edition, Volume 1*.

Under TSO, you can use the SAS/C PUTENV command to set an environment variable with the specified scope. Under CMS, you must use the CMS GLOBALV command. In the OpenEdition shell, use the **export** command.

For example, these commands set the **_DB_COMM** environment variable to TCPIP in the different environments. For TSO and CMS, the variable is defined with an external scope and will remain defined for the life of the session.

(TSO)

```
PUTENV _DB_COMM=TCPIP EXTERNAL
```

(CMS)

```
GLOBALV SELECT CENVSETS _DB_COMM TCPIP
```

(OpenEdition Shell)

```
export_DB_COMM=TCPIP
```

For details on the PUTENV command, see Chapter 8, "Executing C Programs," in *SAS/C Compiler and Library User's Guide, Fourth Edition*. For details, on the GLOBALV command, see the IBM publication *VM/ESA CMS Command Reference* (SC24-5461) for VM/ESA.

Optionally, you can set the environment variables on the command line during independent start-up of the program being debugged. This is equivalent to setting the environment variables with a program scope. For example, under CMS, this command starts MYPGM in debug mode and sets the **_DB_COMM** environment variable to TCPIP:

```
MYPGM =d =_DB_COMM=TCPIP
```

Under MVS batch, you must set the environment variables in an EXEC PARM string. There is a 100 character limit on EXEC PARM strings, so you may need to use another method for setting the environment variables if this presents a problem. For example, you could use *argument redirection* to obtain the environment variables from a file. For details on argument redirection, refer to Chapter 9, "Run-Time Argument Processing," in *SAS/C Compiler and Library User's Guide, Fourth Edition*.

Because CICS does not let you specify program scope variables on the command line, you should use the remote debugger's CICS front-end transaction to set the environment variables for the remote debugging session. See "Debugging CICS Applications" on page 75 for details.

Using the SASCDBG Debugger Interface

The **SASCDBG** debugger interface has three forms: a TSO form, a CMS form, and an OpenEdition shell form. For TSO and OpenEdition, there is also a form for the independent start-up method and a form for the automatic start-up method. The start-up method depends on the presence or absence of a program name. CMS supports the independent start-up method only (no program name on the command line).

Process invocation

When you start the remote debugger under TSO or the OpenEdition shell, you can specify which method you want to use to invoke the program being debugged. Your choices are:

- **fork**
- **oeattach**
- **ATTACH**

The **fork** method is similar to a UNIX fork. It is the standard method in the OpenEdition environment for creating a child process with its own address space. You will typically use the **fork** method when debugging an OpenEdition application under TSO.

The **oeattach** method provides an alternative method for debugging OpenEdition applications under TSO or the OpenEdition shell. It provides OpenEdition-style file-handling, signal-handling, and so forth, but uses the same address space as the debugger. You will typically use the **oeattach** method for improved performance over **fork**.

The **ATTACH** method starts the debugger processes using the assembler **ATTACH** macro. You can use the **ATTACH** method for debugging non-OpenEdition applications under TSO. However, we do not recommend using this method, since the local debugger is considerably more efficient in this environment.

Program I/O handling

When debugging an OpenEdition application under TSO using the **fork** or **oeattach** method, an OpenEdition terminal (tty) is not defined for the program. Instead, the debugger intercepts program I/O to the standard OpenEdition file descriptors 0, 1, and 2, and processes it according to the specifications in the Termin and Termout windows (if the debugger is running in full-screen mode). Any attempt by the program to open **/dev/tty** will fail. If

you invoke the program with **oeattach**, the program can open the file **//ddn:*** to access the TSO terminal.

You can disable program I/O intercepts with the **SASCDBG** DBTERM parameter. This causes program output to the OpenEdition file descriptors to be discarded. In addition, input requests receive an immediate end-of-file. For more information about DBTERM, see section "SASCDBG arguments" on page 73.

SASCDBG syntax

This section describes the syntax for the **SASCDBG** debugger interface. The syntax is different in each environment, but the arguments have the same meaning.

Note: You can abbreviate command keywords to four or fewer characters in most cases: PARMS and RESTART can be abbreviated to a single character; DBCOMM, DBPORT, and DBINVK can be abbreviated to three characters; DEBUG can be abbreviated to two characters; and DBTERM can be abbreviated to four characters. **YES** and **NO** values can be abbreviated to one character. All intermediate abbreviations are accepted. DBLU and DBTP cannot be abbreviated △

TSO Forms:

SASCDBG *pgm_name*
[PARMS(*pgm_args...*)]
[DBCOMM(TCPIP | TCPIP_ *xxx*)]
[DBPORT(*port_num*)]
[RESTART(YES | NO)]
[DBTERM(YES | NO)]
[DBINVK(FORK | OEATTACH | ATTACH)]
[DEBUG(YES | NO)]

SASCDBG
[DBCOMM(TCPIP | TCPIP_ *xxx* | APPC)]
[DBPORT(*port_num*)]
[RESTART(YES | NO)]
[DBLU(*lu_name*)]
[DBTP(*tp_name*)]

CMS Form:

SASCDBG (TCPIP | TCPIP_ *xxx*
[PORT= *port_num*]
[RESTART])

OpenEdition Shell Forms:

sascdbg
[-tcpip | -tcpip_ *xxx*]
[-port= *port_num*]
[-nod]
[-fork | -oeattach]
pgm_name [*pgm_args...*]

sascdbg
[-tcpip | -tcpip_ *xxx*]
[-port= *port_num*]

SASCDBG arguments

All arguments except *pgm_name* and *pgm_args* are case insensitive.

FORK
Requests program invocation with a **fork** function call and then an **exec** of the program to be debugged in the child process of the **fork**. This is the default program invocation method when debugging an OpenEdition application under TSO or the OpenEdition shell.

OEATTACH
Requests program invocation with an **oeattach** function call for improved performance over the **fork** method when debugging an OpenEdition application under TSO or the OpenEdition shell.

ATTACH
Requests program invocation with the MVS **ATTACH** macro. You can use this option to debug non-OpenEdition applications under TSO. However, we recommend using the local debugger instead of the **ATTACH** option.

DEBUG
Determines whether the **=debug** runtime option will be added during program invocation; the default is **YES**. Normally, the **=debug** option must be passed by **SASCDBG** in order to cause the client program to invoke the debugger interface. However, if the program uses the **_nlibopt** external variable to suppress runtime option handling, **=debug** will be interpreted as a program argument. If your program sets **_nlibopt**, it should also initialize the library external variable **_options** to specify **_DEBUG** in order to allow the program to be debugged. For more information about the **_nlibopt** and **_options** variables, refer to Chapter 9, "Run-Time Argument Processing," in *SAS/C Compiler and Library User's Guide, Fourth Edition*.

NOD
Suppresses insertion of the **=debug** runtime option during program invocation. **-nod** is the OpenEdition-equivalent of **DEBUG(NO)**.

DBTERM
Determines whether the debugger will intercept terminal I/O when debugging an OpenEdition application under TSO using the **fork** or **oeattach** method; the default is **YES**.

Note: This option has no effect when debugging a non-OpenEdition program. Terminal I/O is controlled by the initial configuration file settings and the debugger's terminal I/O window intercept settings. △

TCPIP
TCPIP_*xxx*
APPC
Determines the communication method between the debugger processes, either TCP/IP or APPC. When TCPIP is specified, the debugger uses your site's default TCP/IP implementation of non-integrated sockets. When **TCPIP_***xxx* is specified, the debugger uses the TCP/IP implementation specified by the **setsockimp("***xxx* **")** function call. For example, TCPIP_OE requests OpenEdition integrated sockets. You can specify any TCP/IP implementation that is installed and available

on your system. For more information, see the description of the **setsockimp** function in *SAS/C Library Reference, Third Edition, Volume 2*.

Specifying the communications access method on the command line sets the program scope **_DB_COMM** environment variable. If this variable is already defined with an external or permanent scope, the command line specification takes precedence. If you do not specify the communications access method, the **_DB_COMM** environment variable is used.

RESTART
(TSO and CMS only) Determines whether the debugger display process will be restarted after program termination or loss of communication with the client program. The default is **NO**. This option is intended for use with the independent start-up method when you may need to restart the debugger processes, for example, when debugging a pseudo-conversational CICS transaction (see "Debugging CICS Applications" on page 75 for details; also see the *port_num* option).

If you specify the RESTART option, you must use attention (PA1) on TSO or the HX command on CMS to terminate **SASCDBG**. Otherwise, it will continue trying to reconnect with the client program.

Note: This option does not preserve breakpoints or other debugger session parameters. It recreates the debugger's entire C environment and reloads the debugger load modules. △

port_num
(TCP/IP communications access method only) Specifies the TCP/IP port number to be used by the remote debugger. If the debugger cannot use the specified port, for example, because it is in use by another process, it displays a warning message and allows the system to assign a port number. Port numbers can be in the range of 0-65535. If *port_num* is **0**, the system assigns a port number from 1-65535.

If the operating system supports external scope environment variables, the debugger saves the port number from the current session in an external scope **_DB_PORT** variable. (See note for information about environments that do not support external scope variables.) By default, the debugger will try to reuse the port in the **_DB_PORT** variable if you do not set *port_num* explicitly. This behavior is especially useful when you specify the **RESTART** option, for example, when debugging a pseudo-conversational CICS transaction. This allows the client program to automatically re-establish communication with the debugger on subsequent invocations.

The system always assigns the port number if this is the first debugging session since logging in, the **_DB_PORT** environment variable is cleared, or the debugger is unable to reuse the last port number.

Note: Environments such as MVS batch and the OpenEdition shell do not support external class environment variables. In these environments, the debugger uses the specified *port_num* for the current debugging session, but does not retain it for future sessions. △

lu_name
(APPC communications access method only) Specifies the APPC Logical Unit (LU) where the debugger display will be run. The LU name consists of 1-8 nonblank uppercase letters or numbers (A-Z, 0-9). During independent start-up, the LU name is displayed at the terminal and is used by the client program to request an APPC connection to the debugger display. The default value is 8 blanks, which means to use the system base LU.

Note: The default value may or may not work, depending on the APPC definitions for your system. △

tp_name
(APPC communications access method only) Specifies the APPC Transaction Program (TP) name to be registered with the APPC LU. The TP name consists of 1-64 upper- or lowercase alphabetic characters, numbers, or special characters, except $, #, and @. During independent start-up, the TP name is displayed at the terminal and is used by the client program to request an APPC connection to the debugger display. The default value is **SASCDBG**.

pgm_name
Specifies the name of the program to be debugged. This program will automatically connect with the remote debugger display. Under TSO, the program name must be the first argument. Under the OpenEdition shell, specify the program name last. For details on how the debugger locates programs in the OpenEdition hierarchical file system (HFS), see "Pathname resolution under OpenEdition" on page 74.

pgm_args
Specifies one or more runtime arguments to be passed to the program being debugged. The remote debugger adds the **=debug** runtime option automatically unless you specify **-nod** (OpenEdition) or **DEBUG(NO)** (TSO/CMS).

Pathname resolution under OpenEdition

When debugging an OpenEdition program, *pgm_name* takes the form of a pathname in the OpenEdition hierarchical file system. If *pgm_name* does not begin with a **/** (slash), the debugger searches for the program in the following way:

1. If you are running the debugger display in the OpenEdition shell, it first checks your current working directory.
2. Otherwise, it checks your home directory.
3. Finally, it checks the directories on your search path, as defined in the PATH environment variable.

If *pgm_name* begins with a **/** (an absolute pathname), the debugger looks for the program at the specified pathname.

Restrictions

The remote debugger has the following restrictions on its use and operation:

- The automatic start-up method does not support APPC. This is an APPC limitation for programs that use the APPC **Register_Test** facility. See Appendix 1, "APPC Setup for the Remote Debugger," for additional information.
- When debugging an OpenEdition program under TSO, you must use the TCP/IP communications access method if you launched the program with **ATTACH** or **oeattach**. See Appendix 1, "APPC Setup for the Remote Debugger," for more about this restriction.
- When debugging a CMS program from TSO, you can use the APPC communications access method only if you set the debugger environment variables as described in Appendix 1, section "Variable Settings" on page 93.
- Under CMS and OpenEdition, the debugger display component supports only the TCP/IP communications access method.
- When using the **ATTACH** method of program invocation, the debugger environment variables are appended to the program runtime arguments in the form of **=name=value**. This may cause problems for programs compiled with the **_nlibopts** variable, since the environment variables will be interpreted as program arguments, and will not be set as environment variables.

Debugging CICS Applications

CICS does not support a command line. When you start a CICS program, you specify only the program's transaction name; you cannot specify runtime options, environment variables, or command parameters. Since the remote debugger uses environment variables to exchange information with the client program, you must use the remote debugger's CICS front-end transaction to set these variables and launch the program to be debugged.

To use the remote debugger with a CICS application, follow these steps:

1 Start the remote debugger on the system where you do your development, for example:

```
SASCDBG DBCOMM(TCPIP)

SAS/C Remote Debugger can be reached via:
_DB_COMM=TCPIP connect to _DB_HOST=10.1.1.1
  (MVS), at _DB_PORT=13227
```

2 On your CICS system, start the remote debugger's front-end transaction. The transaction name, as distributed with the SAS/C product, is **DBUG**. Its format is:

```
DBUG trans_name
```

Where *trans_name* is the 4-character transaction name associated with the program to be debugged. For example:

```
DBUG ctim
```

Note: Your CICS systems administrator can change the name of the **DBUG** transaction, if desired. Check with your administrator if you have any questions about the name of the transaction at your site. △

Figure 6.2 on page 75 shows the screen displayed by the **DBUG** transaction. You can TAB from field to field and specify values for the debugger environment variables. You can also specify runtime options; the **=debug** option is specified by default.

Figure 6.2 DBUG Transaction Screen

```
                          SAS/C Debugger Front End

Transaction ID: ____ Program name: ________

run time arguments ===> =debug

                              Debugger Environment Variables

_db_comm ===>       (communication access method: TCPIP | APPC)

TCP/IP only variables

   _db_host ===>            (dotted decimal IP address)
   _db_port ===>            (port number of debugger display process)

APPC only variables

   _db_lu   ===>          (SNA LU name where debugger is running)
   _db_tp   ===>          (Transaction Program name of the debugger)
   _db_mode ===>          (logon mode table name)
```

Once you enter the appropriate values, press the ENTER key to start your application. The **DBUG** transaction verifies that the named transaction exists, that the specified program name is associated with that transaction, and that the program can be loaded into memory. If any of these checks fail, the **DBUG** transaction displays an error message and exits. Otherwise, it starts the application, and the next thing you should see is remote debugger display with your application's source code in the Source window.

If you specify the wrong value for any debugger environment variable (for example, the wrong TCP/IP port number), the application will exit with an **ABEND 1219**. You can rerun the **DBUG** transaction and correct the value without restarting the debugger. The **DBUG** transaction saves the latest value of each field in an external scope environment variable. You will see this value when the transaction screen reappears. Simply replace the value with a new one and press ENTER when you are done.

If you are debugging a pseudo-conversational CICS application, we recommend starting the remote debugger with the RESTART parameter under TSO or CMS. The RESTART parameter causes the debugger to restart after program termination with the same connection parameters. If you are using the TCP/IP communications access method, you should not set the **DBPORT** parameter to **0**, since this will force the system to select a new port number when the debugger is restarted. If this occurs, the CICS application will terminate with an **ABEND 1219**. Otherwise, your pseudo-conversational program will reestablish

communication with the debugger automatically, and you can continue debugging your program without interruption.

If the debugger cannot reuse the previous TCP/IP port, for example, because it is in use by another process, it will display its normal connection message, and the CICS application will terminate with an **ABEND 1219**.

Note: CICS applications cannot communicate with the remote debugger over TCP/IP until you start TCP/IP for your CICS region. For IBM TCP/IP, if the start-up transaction has not been executed, your CICS application will terminate with an **ABEND AEY9**. The CICS administrator can start IBM TCP/IP with the IBM CSKE transaction. Other TCP/IP vendors may have different start-up requirements and procedures. △

Startup Scenarios

This section shows examples of starting the remote debugger in different environments.

TSO independent start-up (TCP/IP)

TSO session 1:

```
SASCDBG DBCOMM(TCPIP)
```

TSO session 2:

```
CALL XYZ.LOAD(MYPGM) '=D =_DB_COMM=TCPIP
  =_DB_HOST=MVS =_DB_PORT=1234'
```

Start the remote debugger using the TCP/IP communications access method. Then, using the information displayed at debugger start-up, call the program load module **mypgm** in debug mode, specifying the communications access method environment variables on the command line.

TSO independent start-up (APPC)

TSO session 1:

```
SASCDBG DBCOMM(APPC) DBLU(C02SESS)
  DBTP(SASCDBG)
```

TSO session 2* :

```
CALL XYZ.LOAD(MYPGM) '=D =_DB_COMM=APPC
  =_DB_LU=C02SESS =_DB_TP=SASCDBG'
```

Start the remote debugger using the APPC communications access method, specifying the Logical Unit name and Transaction Program name on the command line (required if the corresponding environment variables are undefined). Then, call the program load module MYPGM in debug mode, specifying the APPC communications access method environment variables on the command line.

Note: The **_DB_TP** environment variable defaults to **SASCDBG**, and is shown in this example for illustration only. △

CMS independent start-up

CMS session 1:

```
SASCDBG (TCPIP
```

CMS session 2:

```
MYPGM =D =DB_COMM=TCPIP
              =_DB_HOST=VM
              =_DB_PORT=1234
```

Start the remote debugger using the TCP/IP communications access method. Then, using the information displayed at debugger start-up, run MYPGM in debug mode, specifying the communications access method environment variables on the command line.

MVS batch independent start-up

TSO session:

```
SASCDBG DBCOMM(TCPIP)
```

MVS batch JCL:

```
//STEP1 EXEC PGM=MYPGM,
//  PARM='=D =_DB_COMM=TCPIP =_DB_HOST=MVS
//            =_DB_PORT=1234'
//STEPLIB  DD DSN=pgm.load,DISP=SHR
//CTRANS   DD DSN=SASC.LOAD,DISP=SHR
//SYSPRINT DD SYSOUT=A
```

Start the remote debugger using the TCP/IP communications access method. Then, submit a batch job to run **MYPGM** in debug mode, specifying the values of the communications access method environment variables in a PARM string.

TSO automatic start-up

```
SASCDBG mypgm DBCOMM(TCPIP) DBINVK(FORK)
```

Start the remote debugger using the TCP/IP communications access method and debug the OpenEdition program MTPGM using the **fork** method.

Note: The program name is case sensitive and is the first argument in the command line. △

OpenEdition shell automatic start-up

```
sascdbg -tcpip -fork /dev/r6/mypgm
```

Start the remote debugger using the TCP/IP communications access method and debug the program a **/dev/r6/mypgm** using the **fork** method.

Note: The **fork** method is the default for invoking a program under the OpenEdition shell, and is included in this example for illustration only. △

---

* With APPC, both sessions must have the same effective userid. See Appendix 1, section "APPC Security" on page 94 for details about this requirement.

rsystem Command

The SAS/C Debugger's **rsystem** command is intended for use with the remote debugger. It is similar to the **system** command described in the *SAS/C Debugger User's Guide and Reference, Third Edition*, but allows you to execute an operating system command in the environment of the program being debugged.

The following reference section describes the **rsystem** command.

rsystem

Execute an Operating System Command on a Remote System

ABBREVIATION
rs{ystem}

FORMAT
rsystem *operating-system-command*

DESCRIPTION
The **rsystem** command is intended for use with the remote debugger. It sends the command specified in the argument *operating-system-command* to the environment of the program being debugged. For example, you might use this command to execute a CMS command from TSO when debugging a CMS program from TSO.

The **rsystem** command cannot be followed by another debugger command on the same line. That is, the arguments to the **rsystem** command are assumed to extend to the end of the line, including any semicolons on the line.

EXAMPLES
```
rsystem alloc fi(iforgot)
 da(to.allocate.this.dataset) shr
```
executes the TSO ALLOCATE command.
```
 rsystem access 391 Q
```
executes the CMS ACCESS command.

SYSTEM DEPENDENCIES
See the discussion of the **system** command in *SAS/C Debugger User's Guide and Reference, Fourth Edition* if the remote program is running under TSO or CMS. Command output, if any, generally appears in the remote program's session or log.

rsystem cannot be used when the remote program is executing under OpenEdition or CICS.

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE
The **rsystem** command is not affected by changes in scope.

RETURN CODES SET

Successful:	return code from operating system
Unsuccessful:	-101 or less;
	-103 means command not found;
	-104 means syntax error;
	-110 means the command is not supported in the target environment;

SEE ALSO
system.

Search Lists

This release of the SAS/C Debugger allows you to create *search lists* for specifying the identity and location of files used by the debugger. Search lists are created with the debugger's **set search** command and **set cache** command.

You can use search lists in any environment where you can run the SAS/C Debugger. However, they are particularly useful in the following situations:

- when you develop applications in a cross-development environment, where compilations occur on a UNIX workstation using the SAS/C Cross-Platform Compiler
- when you compile in batch or TSO and debug in the OpenEdition shell.

The following sections describe how to use **set search** and **set cache** to define search lists. While the discussion includes information about using search lists in a cross-development environment, it focuses on the mainframe environment. For details on using search lists in a cross-development environment, see *SAS/C Cross-Platform Compiler and C++ Development System: Usage and Reference, First Edition*.

Input file selection and specification

The SAS/C Debugger provides access to information from several different types of files, including:

- debugger files
- source files
- alternate source files
- system include files
- user include files

When the default search procedure for a file does not meet your needs, it is possible to change this behavior by using the debugger's **set search** command.

The **set search** command is used to specify *filename templates*. Filename templates are used 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. So, when necessary, 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 patterned after the format argument of the C **printf** function. Each filename template can contain *conversion specifiers* and characters. A conversion specifier is a character or a string preceded by the 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 the open fails the next filename template is processed until either a file is opened or no more filename templates are 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.

Note: If you run the debugger under the OpenEdition shell, the filename string is interpreted as an MVS filename, not as a POSIX filename. If you want to specify searching for a file in the OpenEdition hierarchical file system, you must begin the filename with the **hfs:** filename style prefix. △

Using the set Command

The SAS/C Debugger's **set** command provides two subcommands:

- **set search**
- **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 primarily in a cross-development environment, especially for large debugger files. This command uses a filename template to specify a location where debugger files are searched for, before the debugger uses the debug search list. In a typical cross-debugging session, this cache location would be on the mainframe.

The debugger first looks for the debugger file in the cache location. If the debugger file is found in the cache location, it is used, unless the module has been recompiled since the debugger file was generated. If the module has been recompiled, or if the debugger file is not found in the cache location, the debugger uses the normal search mechanism to locate the file, and then copies the file to the cache location.

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. 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 filename the compiler used to open the file when it was created.

On MVS, if you have not specified a cache location or search list for the debugger file, the debugger uses a default search list for debugger files, which is equivalent to the following command:

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

The debugger will attempt to open the file by the name the compiler used when it created the file if the file is not found using the search list, or if the debug search list was cleared with a **set search debug** command of the form:

```
set search debug = ""
```

Note: If the load module being debugged was generated from objects created by the SAS/C Cross-Platform Compiler, the debugger uses the **path:** filename style prefix with the original filename. △

On CMS, there is no default debug search list. If the debugger file is not found using a search list defined by you, then the debugger will search for the file using its original filename. If the debugger is running under CMS Release 6 or later, it may search one or more shared file directories for the debugger file: If the **_DB** environment variable is set, each *dirname* or NAMEDEF specified using the **_DB** environment variable is searched.

If the debugger file is not found in a Shared File System directory, or you are using a CMS release earlier than 6, the debugger will try to open file *fname* DB *, where *fname* is the filename used by the compiler when it created the debugger file.

To append templates to the debug search list, use the following form of the **set search** command:

```
set search debug + "template1"
  "template2" ...
```

To replace all templates in the debug search list, use the following form of the **set search** command:

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

Locating source files

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

1. any locations specified in the source search list, as specified by the **set search** command
2. the filename the compiler used to open the file when it was created.

On MVS, 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.

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.

You can also 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 files and user include files used to compile your program. The different types of include file 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** command
2. the filename the compiler used to open the file.

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

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

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

set Command Reference

The SAS/C Debugger's **set** command is best used in the debugger PROFILE or run command file 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 **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 definition of search templates that are used to access debugger and source files. 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.

search SUBCOMMAND

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

Format 1:

set search *file-tag* **=|+|-** "*template1* "
["*template2* " **...]**

Format 2:

set search *file-tag* **=**

Format 3:

set search *file-tag*| *** ?**

Format 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:

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 DESCRIPTIONS

Format 1:
This format of the **set** command specifies a search list for the type of files designated by *file-tag*. A search list consists of one or more templates that

are used by the debugger when it needs to locate debugger or source files.

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

= 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:

%lower or **%l**
causes the replacement text for the conversion specifier immediately 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 immediately 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 or SNAME should always be specified when the program is 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 used by the compiler when it opened the file, including any SAS/C style prefix the compiler used. The format of the filename is operating system dependent. This conversion specifier may be difficult to use without a complete knowledge of SAS/C style prefixes and how the compiler derives specific file names. 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 after the last slash, if present. If there is no slash, it is the entire filename the compiler used to open the file.

%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.

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

The filenames generated by the application of the conversion specifiers in the template are passed to the **fopen** function, in an attempt to open the specified file.

For example, on MVS the following *template* would access a PDS member that matches **%sname**:

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

A file in the POSIX-conforming hierarchical file system, as implemented by OpenEdition MVS, could be accessed by a template like this:

```
"hfs:dbgfiledir/%leafname"
```

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

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

A similar *template* could be specified to access source files on CMS in the Shared File System, for example:

```
"sf:%sname C fpool:userid.proj4"
```

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 last 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 all file types supported by **set search**.

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 UNIX 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 that 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 UNIX workstation, this template is used to generate a filename and open the file on the workstation.

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 MVS **dsn:** style filename that is searched if the user include file is not found on the workstation.

set search source = "hfs:/home/cxx/src/%leafname"
specifies a search list for sources files in the OpenEdition hierarchical file system. When the debugger looks for source code, this template is used to generate a filename and open the file.

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.

SYSTEM DEPENDENCIES

The filenames generated by the search templates are operating system dependent.

COMMAND CAN BE ISSUED FROM

PROFILE	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**

Run Command File

This release of the SAS/C Debugger allows you to use a *run command file* to execute debugger commands at startup. The run command file is similar to the debugger's PROFILE, but is intended for use in environments that do not support REXX or CLIST, such as MVS batch. The debugger runs the commands in the run command file if it cannot find an executable startup PROFILE.

The following is a description of how the debugger searches for the startup files in each environment.

- MVS

  If the debugger is running under TSO, it first checks for the PROFILE in DDname DBGPROF. If it cannot open DBGPROF, or if DBGPROF is not allocated, the debugger checks for the PROFILE in the data set *userid*.CDEBUG.CLIST. If it cannot find or open a startup PROFILE, the debugger issues a warning message and begins checking for the run command file.

  The debugger first checks for the run command file in DDname DBGRC. If it cannot open DBGRC, or if DBGRC is not allocated, the debugger checks for the run command file in the data set *userid*.CDEBUG.RC. If the debugger cannot find or open a run command file, a warning message is issued.
- OpenEdition shell

  Under OpenEdition, the PROFILE is an executable file named **.cdebug**. The debugger searches for the PROFILE in the directories named in the PATH environment variable and uses the first executable file it finds named **.cdebug**. If the debugger cannot find a PROFILE, it checks for a run command file at:

  ```
  //hfs:initial_working_directory/.cdebug
  ```

  If the debugger cannot find or open a PROFILE or run command file, a warning message is issued.
- CMS

The debugger first checks for the PROFILE in:

```
//cms:profile cdebug *
```

If it cannot find the PROFILE, it checks for the run command file in:

```
//cms:cdebug rc *
```

If the debugger cannot find or open a PROFILE or run command file, a warning message is issued.

Minor Changes and Enhancements

The following is a list of the minor changes and enhancements in Release 6.00 of the SAS/C Debugger.

- When running the debugger under OpenEdition, you can store debugger startup files in the OpenEdition hierarchical file system. You can also call other REXX scripts stored in the hierarchical file system using the debugger **exec** command. To run the debugger PROFILE or other REXX scripts, these files must have execute permission.

  Note: OpenEdition REXX scripts executed under the debugger should use the REXX statement **address CDEBUG** before issuing any debugger commands. △

- In previous debugger releases, you could allocate the MVS partitioned dataset DBGSRC at runtime. This feature has been dropped due to the availability of the more general **set search source** facility.
- Under OpenEdition, you can use the debugger's **system** command to execute a shell command. **system** will interpret the command string as a shell command.

CHAPTER 7

SAS/C Debugger Changes in Release 5.50

Introduction

This chapter provides a complete description of the changes and enhancements made to the SAS/C Debugger for Release 5.50.

Release 5.50 Enhancements

The following major enhancements to the SAS/C Debugger have been implemented with Release 5.50:

- source level debugging of C++ programs
- line number breakpoints for functions located in an include file
- Browse and Find windows
- **browse**, **window find**, and **log** commands.

In addition to these enhancements, there are several minor enhancements that have been implemented as described in section "Minor Enhancements" on page 86.

Incompatibility with Previous Releases

The SAS/C Debugger, Release 5.50, has been redesigned to support the debugging of programs that have been developed with the SAS/C C++ Development System. To accommodate this change, the debugging information produced by Release 5.50 of the SAS/C Compiler has also been redesigned to support C++. Because of these changes, Release 5.50 of the SAS/C Debugger is not compatible with debugging files generated by any releases of the compiler prior to Release 5.50.

Change in Breakpoint Implementation

The SAS/C Debugger now implements line number breakpoints for functions that are located in an include file. This enables you to use line numbers as the *hook-type* command argument to specify a breakpoint in a function that is in an include file and not the primary file. The following commands are affected by this change:

break	**goto**	**resume**
disable	**ignore**	**runto**
drop	**on**	**trace**
enable	**query**	

Breakpoints in Include Files

Prior to Release 5.50, the debugger supported a variety of hooks whose *hook-type* arguments were the line numbers of the compilation. When using these arguments, you interacted with the debugger in terms of these compilation or primary file line numbers. If a function was in an include file, all lines of the function corresponded to the same primary file line; the granularity provided by the existing mechanism was not fine enough to set a breakpoint on a particular line of the function in the included file.

With Release 5.50, this behavior has changed: line numbers specified as *hook-type* arguments now correspond to the line numbers of the file that contains the function. For functions that are in the primary file of the compi-

lation, you can set breakpoints as before. For functions in include files, the line numbers used are the actual line numbers of the included file that contains the function and not the line numbers of the primary file. When the debugger stops in such an include file, the source for the include file is displayed in the Source window, and the line numbers displayed are those of the included file. This behavior is supported only if the entire function is in one included file, which is normally the case.

Breakpoints by compilation name work as before. For example, the following **`break`** command format sets breakpoints at all hooks in the primary file specified by the *section-name* argument that are on the line specified by the *line-num* argument.

`break` (*section-name*) *line-num*

If that line contains an include file with code, the debugger stops at all line-hooks in that include file, as well as any other files that are included.

Source in Include Files

Under MVS, to display source code in system include files, the DDname DBGSLIB must be allocated to the partitioned data sets containing the include files. For user include files, the same allocations issued at compiler time are required. For instance, if the program specified **`#include "decl.h"`**, the DDname H must be allocated during debugging to the same PDS as at compile time.

Prefix-area commands, which are entered in the prefix area of the Source window, work with include files.

Breakpoint Verification

With Release 5.50, the SAS/C Debugger verifies that there are line-hooks at the location specified by the following formats of the **`break`** command:

`break` *function-name line-num*

`break` *function-name line-num1 :line-num2*

The verification of line-hooks affects the debugger in the following ways:

- If the function is active (either the function was stopped in, or is in the calling sequence), the verification is done immediately. Otherwise, verification is deferred. The debugger stops at the prolog of the function, then verifies and installs the breakpoint.
- If there is no line hook at the line requested, you are given the opportunity to reenter the breakpoint. If a range of lines is specified by a *num1 :line-num2* argument, debugger line-hooks should be present for both lines specified.
- If code for a line is not contiguous and a request by line number is issued, the debugger only uses the hooks in the first contiguous portion. For example, the code for the header of a **`for`** loop with a non-null body is not contiguous. After the code for the initialization and the code for the test, the code for the loop body appears, and only then does the code for the increment portion of the loop appear. Therefore, if the entire loop header appears on one line, setting a breakpoint on that line does not cause the debugger to break before the increment portion.

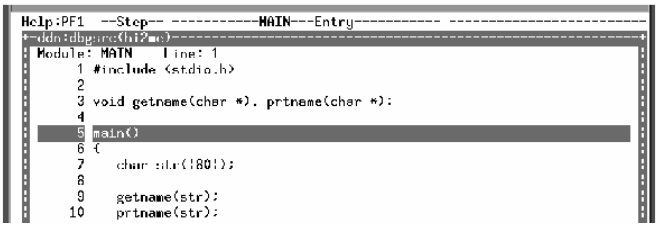
where and transfer Commands

The line numbers displayed in the output of the **`where`** command are primary file line numbers. Similarly, the line numbers returned by the **`transfer`** command are also primary file line numbers.

Browse Window

The Browse window, shown in Figure 7.1 on page 84, is used to browse text files and to display the output of the **`browse`** command.

Figure 7.1 Browse window



The Browse window has the following characteristics:

- The window border is optional.
- The top line contains the name of the displayed file and the next line contains the number of the top-most line displayed in the window.
- The lower portion of the window contains two areas that display the line number of the file being displayed and the text of the file.
- The text area is 252 characters wide, only a portion of which is visible at one time.
- The maximum text file line length supported in full-screen mode is 252.
- The amount of memory used for browse buffers is controlled through the **`window memory`** command or the Config window. If the value of the amount of memory for buffers is changed, any Browse windows opened after the change will have the new value.

Opening a Browse Window

The following command opens a Browse window:

`window open browse`

As many as six Browse windows can be open simultaneously.

You can also use the **`browse`** command to open up a Browse window. See section "browse Command" on page 85 for more information.

Filename Syntax

If an explicit style precedes the filename specified in the File: field of the Browse window, the debugger uses the style specified. If not, the debugger assumes a **tso:** style filename under MVS and a **cms:** style filename under CMS. See *SAS/C Library Reference, Third Edition, Volume 1*, for more information about filename specifications.

Moving Around the Text File

Scrolling vertically using the **window scroll up** and **window scroll down** commands moves the data in both the line-number and the text areas of the Browse window. Scrolling horizontally using the **window scroll left** or **window scroll right** commands moves only the text and not the line numbers.

Scrolling is a simple and fast way to move to lines that are close to the lines that are currently displayed. A faster way to move to distant parts of a text file is to enter the desired line number in the Line: field of the Browse window.

A different text file can be viewed in the Browse window by entering its filename in the File: field.

If you enter either an invalid filename in the File: field or an invalid line number in the Line: field, a pop-up window opens that allows you to correct the invalid input.

The **window find** command, which is described later in this chapter, is also supported in the Browse window.

Order of Processing

If input is specified in more than one field of the Browse window, the data are processed in the following order:

1 File: field
2 Line: field

New Commands

Three new commands have been introduced with Release 5.50: the **browse** and **log** commands and the **window find** subcommand.

browse Command

The **browse** command is used to browse the area of the source file where the *name* being browsed is declared. The format of the **browse** command is as follows:

browse [struct | union | class | enum] *name*

The *name* argument is a single identifier name, not an expression.

The class keyword is valid only if **auto cxx** is in effect. The **cxx** keyword is set automatically whenever the debugger detects C++ translated source code.

If the optional struct, union, class, or enum keyword is not specified, the debugger performs a search in the following order:

1 the list of preprocessor symbols, if present.
2 the list of identifiers, typedefs, and enumeration constants.
3 the list of struct, union, enum, or class tag names.

If one of these optional keywords is specified, only the list of tag names is searched.

Normal C scope rules apply to all searches; command scope is used. If a declaration for the name is found, a Browse window is opened on the file and positioned to the line containing the declaration.

The **browse** command may be preceded with a **>** or **>>** command prefix: a **>** opens a new Browse window; a **>>** or no prefix reuses the most recently used Browse window or opens one if none is open.

Note: The only way to issue the **browse** command is through the Command window (or a PF key). You cannot issue the **browse** command in the Browse window. △

log Command

The **log** command can be used to log the contents of the Log window to a dataset. The **log** command takes the following forms:

Format 1:
log file [*filename*]

Format 2:
log append *filename*

Format 3:
log start | stop | capture

The file keyword specifies the file to which logged output is to be written. The **log** command writes over the file. If the **log file** command is issued without any arguments, the name of the current log file is displayed.

The append keyword specifies the file to which logged output is appended.

The *filename* argument is specified as a **tso:** style filename under MVS, and a **cms:** style filename under CMS. Do not, however, specify the **tso:** or **cms:** prefix in the command; it is assumed.

Issuing the **log** command with either a file or an append keyword and a *filename* argument specifies the file to be used for logging. However, it does not start the logging process. Logging of the contents is started by issuing a **log start** command. Logging is turned off by issuing a **log stop** command. The **log stop** command does not close the file; it flushes the file to disk. Logging may be resumed at anytime by another **log start** command.

Issuing a subsequent **log file** *filename* or **log append** *filename* command closes the current log file and opens the file specified for logging.

The **log capture** command is used to log everything in the debugger's Log window buffers since the last **log stop**. Some log output may be lost if the Log window buffer is not large enough.

window find Subcommand

With Release 5.50, a new **window** subcommand, **find**, has been added. The **window find** command is used to search for strings and is supported in the following windows:

- Browse
- Log
- Source

The following format is used with the **window find** command:

window find *window-name*

The *window-name* argument can be any of the following:

- <>
- browse
- log
- source.

If the **<>** *window-name* argument is used, the position of the cursor determines the window that the command is applied to. If a window name is specified as the *window-name* argument, the position of the logical cursor is used to determine the starting point of the search, if the search is cursor dependent.

Find Window

When you execute the **window find** command, the Find window, illustrated in Figure 7.2 on page 86, is opened. If you have not changed your PF key assignments, you can also use PF17, which is assigned to the **window find <>** command, to open the Find window. Once the Find window is open you can enter the string to be searched for and the desired occurrence: **f** (first), **n** (next), or **p** (previous). The search is started when you press the ENTER key or any PF key.

Figure 7.2 Find Window

If **n** or **p** is specified in the Occurrence: field, the search begins from the current position of the logical cursor. If this is the first time the **window find** command is being executed in this window, or if the last time the command was executed was in a different window, occurrence defaults to **P** for the Log window and **n** for the Source and Browse windows.

The search string may contain embedded blanks. By default, searches are not case sensitive. Specifying **Y** in the Case Sensitive: field makes searches case sensitive.

In the current implementation, subsequent occurrences may be found by pressing the PF key assigned to the **window find <>** command (PF17 by default) twice. The first key press opens the Find window with the search string used the last time, and an appropriate occurrence parameter. The second key press causes the debugger to accept the contents of the window and begin searching.

To abort a search after the Find window has been opened, erase the text of the string.

Minor Enhancements, Incompatibilities, and Corrections

This section describes the minor enhancements and incompatibilities in Release 5.50 of the SAS/C Debugger, and errors in the Release 5.50 debugger documentation.

Minor Enhancements

The following minor enhancements to the SAS/C Debugger are introduced with Release 5.50:

- When displaying storage, the debugger now takes program AMODE into account. If the program is in AMODE 24, the high-order byte of a pointer is ignored when using that pointer to address memory. This purification only takes place when pointers are dereferenced. This means that, for example, the command **dump 0p04c78000** always dumps memory at the address specified, even if the C program is running in AMODE 24. But, for an AMODE 24 program, if a pointer variable **p** contains this value, then the command **dump p** displays the contents of 0pc78000.
- The line number area of the Source window has been widened by one character. It was five characters wide; now it is six characters wide.
- The maximum value of the **M** argument of the **window context source N M** command is now 0x7fffffff.
- The **list** command now supports line numbers greater than 64K.
- The format of the output of the **print** command when directed to the Log window is the same as when directed to the Print window. Therefore, the command **print** *expression*, where the *expression* argument is the name of an array, prints all the elements of the array. A format, if specified, applies to each individual element, except for character arrays with **s** format specifications.
- Line hooks now support line numbers greater than 65536.
- If a variable of file scope is hidden by a variable of more local scope, the file scope variable can be accessed in any command that supports expressions by using the C++ unary operator **::**. For example:

```
print ::xyz
```

- The **whatis** command can be used to examine the types of class members by using the C++ binary operator **::**. For example:

```
whatis my_struct::member_name
```

- Specifying the struct, union, or enum keyword when using a cast in an expression is optional.
- If you are intercepting output in the Termout window with pause turned on, then at program completion time

if there is any output that you have not seen, the debugger stops to let you view it.

- If you issue an **auto cxx** command, the debugger automatically saves the current status of **auto** command's extname keyword and turns on **auto extname**. When **auto cxx** is turned off, the most recently saved value of extname is restored. **auto extname** cannot be turned off while **auto cxx** is on. The **query** command displays the state of the cxx keyword. The **transfer** command supports the cxx keyword. If all **auto** command settings are transferred, cxx appears between extname and linesize.

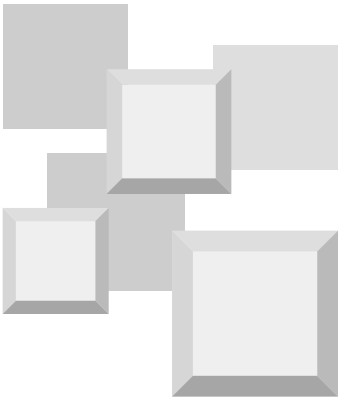
Minor Incompatibilities

The following is a list of incompatibilities between Release 5.50 of the SAS/C Debugger and previous releases:

- The **window memory** command now takes an additional argument representing the amount of memory used for buffers for each invocation of the Browse window. The debugger does not accept old-style **window memory** commands because they are now syntactically invalid. The simplest solution to avoid repetitive warning messages is to save the configuration with a **config save** command the first time the Release 5.50 debugger is invoked. The Config window contains an additional field that is used to specify the amount of memory associated with the Browse window.
- If the extended line-number information is not available for a compilation, function line numbers are parenthesized to represent compilation number.
- The default value of the **auto linesize** command is changed to 75. (It was 78.)
- With the introduction of the cxx keyword to the **auto** command, the minimum lengths you can specify for the cmacros and nocmacros keywords of the **autos** command are now 2 and 4 respectively.
- Support for (PTYPE) in Format 3 of the **print** command has been dropped. With the introduction of casts in expressions, there is no longer a need for this method of casting.
- The *function-name:type* format of the *expression* argument is no longer supported. The **scope** command in Release 5.00 provides similar functionality.
- There are no limits to the length of identifiers that the debugger can process.
- Debugger error output goes to DDname DBGTERM under MVS batch.
- The screen is refreshed after completion of execution of the **system** and **escape** commands.
- The **query** command displays the **ignore signal** command as **ignore** <all signals>.

Minor Corrections

The documentation for the **resume** command should say in the SCOPE section that the command is affected by **scope**.



PART 2

Appendices

APPENDIX

1

APPC Setup for the Remote Debugger

This appendix describes how to configure APPC for use with the SAS/C Remote Debugger. If you are a current APPC user, you may have already performed most of the procedures described here. Otherwise, you will need to modify APPC configuration files and define APPC resources, and should proceed only if you are familiar with IBM system or network administration. No attempt is made to provide all of the details of configuring APPC in each of the environments in which the remote debugger is supported. You should refer to the IBM publications mentioned in this appendix for complete information.

Introduction

The remote debugger consists of two cooperating processes with a client/server relationship: the debugger display and control process and the client program process (see "Architecture" on page 70). The debugger display process can communicate with the client program using IBM's Advanced Program-to-Program Communication (APPC) access method if (and only if) that process is running in a different address space or on a different system.

The debugger display process is the server, and is the inbound side of the APPC conversation. It dynamically registers the debugger's Transaction Program (TP) name through the APPC/MVS **`Register_Test`**, **`Accept_Test`**, and **`Get_Conversation`** service calls. The program being debugged is the client, or outbound side of the APPC conversation, and issues the outbound CPI-C **`Allocate`** call to connect to the debugger display.

Currently, the debugger display process supports APPC under MVS only. The client program can run in any environment that supports the SAS/C runtime library and the APPC Common Programming Interface Communications (CPI-C) API. However, APPC can be used as the communication method only when you start the debugger using the independent startup method. You must use the TCP/IP communications access method whenever you start the debugger processes with the automatic startup method.

The APPC setup requirements are different for each environment in which the remote debugger is supported. The following sections describe these requirements.

MVS Setup Requirements

To use APPC with the remote debugger under MVS, you need to perform the following tasks:

1. Allocate the APPC VSAM-side information and transaction program profile datasets. The APPC software includes example datasets in SYS1.SAMPLIB members ATBSIVSM and ATBTPVSM, respectively. ATBSIVSM defines the SYS1.APPCSI dataset and ATBTPVSM defines the SYS1.APPCTP dataset.
2. Add a logical unit (LU) definition to SYS1.PARMLIB member APPCPM*xx*, where *xx* is a two-character qualifier that distinguishes different members used by different system(s). (The operator or system programmer chooses this qualifier at system initialization time through other SYS1.PARMLIB members.) You can use an existing logical unit if the **`LUADD`** definition includes TPLEVEL(USER), indicating that USER level Transaction Programs are going to be used.

   The following sample definition defines C02SESS as a logical unit:

   ```
   LUADD ACBNAME(C02SESS)
       TPDATA(SYS1.APPCTP) TPLEVEL(USER)
   ```

   To use this LU, you would set the debugger environment variable **`_DB_LU`** to **`C02SESS`**.
3. If the debugger display process and client program are going to run on different systems or LUs on the VTAM network, define a VTAM APPL for the LU to be used by the remote debugger. For example, this APPL definition might appear in SYS1.VTAMLST or an equivalent VTAM definition file:

   ```
             VBUILD TYPE=APPL
   C02SESS   APPL   ACBNAME=C02SESS,APPC=YES,
                    AUTOSES=0,DDRAINL=NALLOW,
                    DMINWNL=10,DMINWNR=10,
                    DRESPL=NALLOW,DSESLIM=10,
   ```

```
MODETAB=ISTINCLM,
SECAPT=ALREADYV,SRBEXIT=YES,
VERIFY=OPTIONAL,VPACING=2
```

The APPL could can then be activated by a VTAM **VARY** command of the form:

```
VARY NET,ACT,ID=C02SESS
```

The specified MODETAB is the system-supplied *logon mode table* for VTAM and can be used with the remote debugger and client program. (In fact, almost any available MODETAB can be used.) However, when the client program is running on a different system, node, or LU on the VTAM network, or is a CICS application, you must specify the mode table with the **_DB_MODE** environment variable.

Configuring a network of workstations and terminals under MVS requires a number of VTAM definitions. VTAM requires definitions for each workstation (physical unit) and the logical units (LUs) through which transaction programs, such as the remote debugger, communicate. For details on VTAM/APPC requirements, refer to the IBM publication *MVS/ESA Planning: APPC Management, Version 4.0* (GC28-1110-0). For details on VTAM definitions, refer to the IBM publications *VTAM Network Implementation Guide, V3/R4* (SC31-6434-1), and *VTAM Resource Definition Reference, V3/R3* (SC31-6412-1).

CICS Setup Requirements

CICS has the same APPC setup requirements as MVS, and these additional requirements:

1 CICS requires an ACF/VTAM APPL definition for its own APPC use.
2 CICS requires a logon mode table named SNASVCMG.
3 CICS requires a SNASVCMG session definition for its LU services manager.

These definitions are reserved for CICS itself and cannot be used by any other applications, including the remote debugger.

In addition to the requirements above, you must also define the following CICS resources using macro-level definitions or the appropriate Resource Definition Online (CEDA) screen:

- The remote debugger LU (**_DB_LU** environment variable) must be defined as a CICS CONNECTION resource.
- The mode table used by the client program to connect to the remote debugger (**_DB_MODE** environment variable) must be defined as a CICS SESSION resource.

Use the CEDA DEFINE CONNECTION screen, shown in Figure A1.1 on page 92, to define the remote debugger LU.

Use the CEDA DEFINE SESSION screen, shown in Figure A1.2 on page 93, to define the client program mode table.

For details on configuring CICS for use with APPC, refer to the IBM publication *CICS/ESA Intercommunication Guide, V3/R3* (SC33-0657-02). Also see *CICS/ESA Resource Definition (Online), V3/R3* (SC33-0666-02).

CMS Setup Requirements

This section describes how to use APPC with the remote debugger for client program communication under CMS. Details on this procedure are documented in the IBM

Figure A1.1 DEFINE CONNECTION screen

```
DEFINE CONNECTION
OVERTYPE TO MODIFY                                        CICS RELEASE = 0321
 CEDA  DEFine
  Connection   ==> SASC
  Group        ==> SASSESS
  DEscription  ==> CICS Connection definition for APPC C02SESS LU
 CONNECTION IDENTIFIERS
  Netname      ==> C02SESS
  INDsys       ==>
 REMOTE ATTRIBUTES
  REMOTESystem ==>
  REMOTEName   ==>
 CONNECTION PROPERTIES
  ACcessmethod ==> Vtam             Vtam | IRc | INdirect | Xm
  Protocol     ==> Appc             Appc | Lu61
  SInglesess   ==> No               No | Yes
  DAtastream   ==> User             User | 3270 | SCs | STrfield | Lms
  RECordformat ==> U                U | Vb
 OPERATIONAL PROPERTIES
  AUtoconnect  ==> All              No | Yes | All
  INService    ==> Yes              Yes | No
 SECURITY
  SEcurityname ==>
  ATtachsec    ==> Local            Local | Identify | Verify | Persistent
                                    | Mixidpe
  BINDPassword ==>                  PASSWORD NOT SPECIFIED
  BINDSecurity ==> No               No | Yes
```

manual *VM/ESA Connectivity Planning, Administration, and Operation* (SC24-5648-01).

Installing the AVS Virtual Machine

To use APPC with the remote debugger under CMS, you need to install the APPC/VM VTAM Support (AVS) virtual machine and make sure that it is activated. You must define and activate one of more APPC AVS gateway LUs for outbound connections, which requires VTAM VBUILD major node definition and APPL definitions. To define the AVS virtual machine to VTAM, you also need a logon mode table for communication on the SNA network with other APPC LUs.

The AVS VTAM APPC LU definitions can be similar to the VTAM APPL definition for MVS. You can also use the AVS default sample logon mode table AGWTAB. However, you may need to modify these definitions to meet the needs of the VM installation.

Once you complete these tasks, you can activate the AVS gateway(s) and set various session limits with AVS commands such as **AGW ACTIVATE GATEWAY** and **AGW CNOS**. You can use the following CMS command to find the currently active gateway LUs for the AVS virtual machine:

```
QUERY GATEWAY ALL
```

To connect the VM system running the client program with the system running the remote debugger display, you may need to define other VTAM SNA physical and logical network connections. For details on VTAM definitions, refer to the IBM manuals *VTAM Network Implementation Guide, V3/R3* (SC31-6434-1) and *VTAM Resource Definition Reference, V3/R3* (SC31-6412-1).

Variable Settings

To use APPC as the communication method when running the client program under CMS and remote debugger display under MVS, the debugger environment variables must be set as follows:

- Both partners must set **_DB_COMM**=APPC.
- On the debugger display (MVS) side, set the **_DB_LU** variable:

  ```
  _DB_LU=remote_debugger_display_LU
  ```

  If necessary, override the default **SASCDBG** Transaction Program name by setting the **_DB_TP** variable:

  ```
  _DB_TP=some_tp_name
  ```

- On the client program (CMS) side, set the **_DB_LU** variable using this special two-part syntax:

Figure A1.2 DEFINE SESSION screen

```
DEFINE SESSION
OVERTYPE TO MODIFY                                        CICS RELEASE = 0321
 CEDA  DEFine
  Sessions     ==> SASSESS
  Group        ==> SASSESS
  DEscription  ==>
 SESSION IDENTIFIERS
  Connection   ==> SASC
  SESSName     ==>
  NETnameq     ==>
  MOdename     ==> ISTINCLM
 SESSION PROPERTIES
  Protocol     ==> Appc              Appc | Lu61
  MAximum      ==> 010 , 010         0-999
  RECEIVEPfx   ==>
  RECEIVECount ==>                   1-999
  SENDPfx      ==>
  SENDCount    ==>                   1-999
  SENDSize     ==> 07680             1-30720
  RECEIVESize  ==> 07680             1-30720
  SESSPriority ==> 000               0-255
  Transaction    :
 OPERATOR DEFAULTS
  OPERId         :
  OPERPriority   : 000               0-255
  OPERRsl        : 0                                          0-24,...
  OPERSecurity   : 1                                          1-64,...
 PRESET SECURITY
  USERId       ==>
 OPERATIONAL PROPERTIES
  Autoconnect  ==> Yes               No | Yes | All
  INservice    ==>                   No | Yes
  Buildchain   ==> Yes               Yes | No
  USERArealen  ==> 000               0-255
  IOarealen    ==> 00000 , 00000     0-32767
  RELreq       ==> No                No | Yes
  DIscreq      ==> Yes               No | Yes
  NEPclass     ==> 000               0-255
 RECOVERY
  RECOVOption  ==> Sysdefault        Sysdefault | Clearconv | Releasesess
                                     | Uncondrel | None
  RECOVNotify  ==> None              None | Message | Transaction
```

```
_DB_LU=AVS_gateway_LU
remote_debugger_display_LU
```

Where *AVS_gateway_LU* is an LU name that defines AVS to VTAM and is separated from the debugger display LU name by a single space. For example, if C09GATEW is the AVS gateway LU and the remote debugger has registered its Transaction Program at LU C02SESS, you could set **_DB_LU** with the following CMS GLOBALV command:

```
GLOBALV SELECT CENV SETLP _DB_LU
C09GATEW C02SESS
```

Note: To specify an environment variable on the command line that includes a space, enclose the value in parentheses, for example, **=DB_LU=(C09GATEW C02SESS)**. △

□ On the client program (CMS) side, set the **_DB_MODE** variable to specify the AVS logon mode table:

```
_DB_MODE=AGWTAB
  (or other VTAM logon mode name)
GLOBALV SELECT CENV SETLP _DB_MODE AGWTAB
```

If you specified a different TP name for the debugger display, override the default **SASCDBG** Transaction Program name by setting the **_DB_TP** variable:

```
_DB_TP=some_tp_name
GLOBALV SELECT CENV SETLP
  _DB_TP some_tp_name
```

Note: If the client program is halted (using CMS HX) or terminates abnormally, the virtual machine may return to the CMS Ready; prompt. To release the APPC connection to the remote debugger display, you may need to issue another command, such as QUERY TIME. After entering the command, the remote debugger display will get an APPC error, release the APPC session, and terminate. △

APPC Security

Since the remote debugger uses CPI-C APPC and user-level transaction programs, effectively, the SECURITY resource equals SAME (SECURITY=SAME) in all environments. This means that whenever you run the remote debugger using APPC, the client program process must have the same effective userid as the debugger display process. For example, if the remote debugger display is running under TSO userid SASCXXX, then to debug a CICS program, the CICS session must be signed on with userid SASCXXX from, the CESN transaction.

Batch jobs and APPC address spaces inherit the effective userid of the submitter, and do not generally pose a security problem. However, to support the case when the client program is running on a different system on the VTAM network, you may need to update certain RACF (or equivalent product) resource classes, including:

□ VTAMAPPL APPCLU
□ APPL
□ APPCPORT APPCTP
□ APPCSI

For details on these resource classes, refer to the IBM publication *MVS/ESA Planning: APPC Management, Version 4.0* (GC28-1110-0).

Note: The TCP/IP communications access method does not require the remote debugger processes to have the same effective userid. Therefore, you may want to use TCP/IP as the remote debugger communication method to avoid the restrictions imposed by APPC. △

APPENDIX

2

Remote Debugger User Exits

Introduction

The remote debugger supports the use of *program invocation exits* for starting applications with unique startup requirements or where automatic startup is unsupported, such as CMS, CICS, or MVS batch. A program invocation exit can be written in C or assembly language and perform any processing that it needs to start the program to be debugged. For example, you might use a program invocation exit to:

- build and submit JCL
- initiate a CICS transaction
- AUTOLOG a CMS session.

The program invocation exit can use the full SAS/C library and any other operating system services available to normal (unauthorized) SAS/C environments.

Calling Sequence

The remote debugger calls the program invocation exit just before it would normally wait for a TCP/IP or APPC connection request from the program to be debugged. When the exit finishes its processing and returns, the debugger waits for the connection request from the remote program and continues initializing once the request is serviced.

The program invocation exit is called once per remote debugger session. It is not called at all when the debugger is running in local mode (for example, when the **`_DB_COMM`** environment variable equals **`LOCAL`** or **`NONE`**).

The remote debugger supports the input and output exits L$UDBIN and L$UDBOUT described in Appendix 3, "Debugger I/O Exits," in *SAS/C Debugger User's Guide and Reference, Third Edition*. The program invocation exit is independent of the I/O exits. The L$UDBOUT output exit is called for initialization before the program invocation exit is called. The program invocation exit and debugger output exit can exchange information using the CRAB user words. This allows the output exit to clean up resources allocated by the invocation exit during the output exit termination call.

Installation Requirements

The program invocation exit takes the form of a separate load module named L$UDBNVK. Under CMS, this load module must be placed in the L$CUSER LOADLIB (created by the user). Under MVS, it may be placed in the transient library data set sasc.LINKLIB, or in a separate library which is concatenated ahead of the transient library to an appropriate DDname (STEPLIB or CTRANS).

Dummy Exit Routines

As installed under MVS, the SAS/C runtime library contains a dummy L$UDBNVK exit routine in sasc.LINKLIB. This routine does nothing; it simply returns control to the caller. You can replace this load module with your own exit or delete it entirely. If you delete the dummy exit, your operating system might produce error messages when the remote debugger is invoked (for example, MVS message CSV003I), but the debugger will start up normally.

Note: There is no dummy exit for CMS. △

Assembly Language Implementation

The program invocation exit is invoked using standard IBM linkage conventions. Any registers used by the routine should be saved (the standard save area addressed by register 13 can be used for this) and restored on exit. You should write the exit so that it can be entered in AMODE 31; however, it always starts in the same addressing mode as the remote debugger's first load module.

When the program invocation exit receives control, register 1 points to the following parameter block.

Offset	Description
0	Address of null-terminated string for the **_DB_COMM** environment variable value. This value indicates the debugger communications access method and will be "APPC", "TCPIP", or the "TCPIP_*xxx*" variant.
4	Address of null-terminated string for the **_DB_HOST** (TCP/IP) or **_DB_LU** (APPC) environment variable value.
8	Address of null-terminated string for the **_DB_PORT** (TCP/IP) or **_DB_TP** (APPC) environment variable value.
12	Address of area mapped by C **struct RDBG_RUNOPTS** or assembler **RDBGOPTS DSECT**.
16	Address of the value from CRABUSR1 (CRAB user word1).
20	Address of the value from CRABUSR2 (CRAB user word2).
24	Address of the value from CRABUSR3 (CRAB user word3).
28	Address of the value from CRABTUSR (CRAB user word4).

Offset 12 addresses the area mapped by the **RDBGOPTS DSECT** shown in Example Code 7.1 on page 96. This DSECT is created from the runtime options specified when **SASCDBG** is invoked, and provides access to the name of the program to be debugged, its runtime arguments, and the program invocation method (**fork**, **oeattach**, **ATTACH**, manual).

Offsets 16, 20, 24, and 28 address CRAB user words one through four, respectively. The CRAB user words are the remote debugger's CRAB, since the program being debugged has not yet received control. You can modify the CRAB user words by replacing the values in the parameter block. When the program invocation exit returns, the new values will be copied into the debugger's CRAB.

You can also modify other information in the **RDBGOPTS DSECT** to affect later debugger invocation processing.

The program invocation exit must return one of the values described in section "Return Codes" on page 97. These return codes are used by exit routines written in both assembly language and C.

C Implementation

You can implement the program invocation exit as a C function with the following syntax:

```
#include <rdbgnvk.h>
int _dynamn (char *rdbg_comm_method,
             char *rdbg_host_lu,
             char *rdbg_port_tp,
             struct RDBG_RUNOPTS *rdbg_runopts,
             void *crabusr1_copy,
             void *crabusr2_copy,
             void *crabusr3_copy,
             void *crabtusr_copy)
        {
          /* exit code */
```

All arguments are input arguments, and are passed by the remote debugger when it calls the exit. The arguments correspond to the information in the parameter block and **RDBGOPTS DSECT** described in the previous section.

The **RDBGOPTS DSECT** maps the C **struct RDBG_RUNOPTS**. This structure is created from the runtime options specified when **SASCDBG** is invoked,

Example Code A2.1 RDBGOPTS DSECT Mapping

```
         SPACE 3
RDBGOPTS DSECT
RDBBPNAM DS    A        Address of program name to be invoked
RDBGARGS DS    A        Address of runtime arguments passed to program
*
         DS    XL2      --Reserved
RDBGOPTF DS    XL1      Option flag
RDBGSUPD EQU   X'01'    Suppress automatic =D runtime arg insertion
RDBGDBTN EQU   X'02'    Intercept OpenEdition Terminal I/O
*
RDBGINVM DS    XL1      Request invocation method for program
RDBGMANU EQU   X'00'    Manual invocation
RDBGFORK EQU   X'01     Via OpenEdition fork and exec services
RDBGOEAT EQU   X'02'    Via OpenEdition/TSO OEATTACH service
RDBGATTA EQU   X'03'    Via MVS ATTACH
*
         DS    0D
RDBGOLEN EQU   *-RDBGOPTS Length of DSECT
```

and provides access to the name of the program to be debugged, its runtime arguments, and the program invocation method (**fork**, **oeattach**, **ATTACH**, manual).

You can modify the information in the **RDBG_RUNOPTS struct** to affect later debugger invocation processing. You can also modify the copies of the CRAB user words. When the program invocation exit returns, the new values will be copied into the debugger's CRAB.

The header file **{tw:regular<rdbgnvk.h>}**, shown in Example Code 7.2 on page 97, is stored in sasc.MACLIBC, and defines the prototype for the program invocation exit and the **RDBG_RUNOPTS** structure. It also defines the return code constants **RDBG_CONTINUE_RC**, **RDBG_SUPRESS_MSG_RC**, **RDBG_SUPRESS_DBGINVOKE_RC**, and **RDBG_FAIL_RC**. The program invocation exit must return one of these values, as described in "Return Codes" on page 97.

Return Codes

The program invocation exit must return one of the following values:

Return Codes

Value	*Meaning*
0	Continue with debugger normal processing and program invocation, as defined by **RDBGOPTS DSECT** or C **struct RDBG_RUNOPTS**. If the program invocation method is manual, issue a message containing process connection information.
4	Continue with debugger normal processing and program invocation, as defined by **RDBGOPTS DSECT** or C **struct RDBG_RUNOPTS**, but suppress the process connection information message.

Example Code A2.2 Remote Debugger Header File

```
#ifndef _RDBGNVK
#define _RDBGNVK
/*
 * This header file defines the prototypes and options struct passed
 * to the L$UDBNVK remote debugger user exit which can be used to
 * start the application to be debugged by the remote debugger.
 */
/* Prototype for exit function */
int l$udbnvk(char *rdbg_comm_method, char *rdbg_host_lu,
             char *rdbg_port_tp, struct RDBG_RUNOPTS *rdbg_runopts,
             void *crabusr1, void *crabusr2, void *crabusr3,
             void *crabtusr);
/* Define return codes for l$udbnvk */
#define RDBG_CONTINUE_RC 0
#define RDBG_SUPRESS_MSG_RC 4
#define RDBG_SUPRESS_DBGINVOKE_RC 8
#define RDBG_FAIL_RC 12
/* Define options struct passed to l$udbnvk exit */
struct RDBG_RUNOPTS
{
 char *pgm_name;    /* Name of program to be invoked                */
 char **pgm_args;   /* Runtime arguments to be passed to program    */
 char optresv[2];
 char optflag;
#define RDBG_SUPPRESS_D 1 /* Suppress auto. =D option insertion     */
#define RDBG_DBTERM_NO  2 /* Turn off Posix Terminal I/O Intercepts */
 char invoke_method; /* Requested invocation method for program     */
#define RDBG_MANUAL   0
#define RDBG_FORK     1
#define RDBG_OEATTACH 2
#define RDBG_ATTACH   3
;
#endif
```

8	Continue with debugger normal processing but suppress debugger program invocation processing and the connection information message.
12	Terminate debugger connection processing and remote debugger initialization and issue a failure message.

Return codes 0 and 12 are typically used as the standard success and failure statuses. Return codes 4 and 8 are intended for special purposes. For example, the exit might return the value 4 after starting a program under MVS batch. It might return the value 8 if your site uses a custom program invocation method.

Index

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Special Characters

Numbers

Your Turn

If you have comments or suggestions about *SAS/C Software: Changes and Enhancements, Release 6.50*, 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 Institute Inc.
Publications Division
SAS Campus Drive
Cary, NC 27513
email: yourturn@unx.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@unx.sas.com