IBM Secureway Cryptographic Products
IBM 4758 PCI Cryptographic Coprocessor
CP/Q Operating System
C Runtime Library Reference
Version 1 Driver 16

Thirteenth Edition, May 20th, 1998

This edition of the C Runtime Library manual is for Driver 16 of the C Runtime Library issued with CP/Q Version 1

Comments or queries concerning this document should be addressed to:

D. C. Toll
IBM Research Division, T. J. Watson Research Center
PO Box 704,
Yorktown Heights
New York 10598, U.S.A.

E-mail: toll@watson.ibm.com
VNET id: TOLL at YKTVMV
☎ (tie-line) 863 7019
☏ (external, USA) 914 784 7019

FAX:
☎ (tie-line) 862 4426
☏ (external, USA) 914 945 4426

This manual was produced using IBM DCF/SCRIPT release 4 and IBM BookMaster release 4.

Contents

Figures

Tables

Things changed in this edition of the manual

Edition 12

- Description added to Introduction chapter describing OS/2 Version 2 as a development environment.
- Added *swab()* function.
- Changed the *pause()* function name to *paws()*.
- Changed resolution of the *clock()* function.
- The function *isPL0()* has been removed. Use *issupv()* instead.
- Added *fabs()*, *pow()*, *log()*, and *log10()* functions.
- C++ support has been added for IBM C Set++ V2.0 and V2.1 compilers (INTEL) as well as IBM xlC V2.1 (POWER/PowerPC). The functions *new()* and *delete()* have been added, and static constructor/destructor support has been added.

Preface

CP/Q is a high-performance 32-bit operating system designed to exploit and work with Intel's 80386 and 80486 microprocessors. One of the major goals of the design has been to provide a small and lean kernel with excellent performance on critical functions such as task switching, message passing and interrupt response time, thereby enabling real-time applications such as communication controllers. An equally important goal that has evolved over time has been to support distributed workstation applications on the same platform. This has meant that we had to have a "pay-as-you-go" design philosophy. In other words, we needed to design a system which could be extended, but not necessarily at the (performance) expense of the basic functions alluded to above. Extra functions (whether inside or outside the kernel) may be slower, but should have little or no impact on core functions.

The first chapter of this document describes C calling conventions, register usage, and the C Runtime Library startup code as well as other language support for the C Runtime Library. The following chapters describe the global variables and function calls available from the C Runtime Library.

Notation

Numeric values are in decimal notation unless explicity stated otherwise, or unless they are in the form 0xnnnn or nnnh, in which case nnnn is in hexadecimal notation.

If a return value is specified as 'nn', it indicates a positive non-zero return value. If a return value is specified as '-nn', it indicates a negative non-zero return value.

Byte ordering

Bytes are numbered and displayed intelligently, that is top to bottom left to right ascending byte addresses.

Chapter 1. Introduction

This manual is a description of the C Runtime Library distributed with the CP/Q Operating System.

The C Runtime Library is intended to support application programmers and is designed to be as close as possible to the American National Standards Institute (ANSI®) X3.159-1989 C standard.

A copy of the ANSI C document can be obained from the following address:

```
Address:
American National Standards Institute
1430 Broadway
New York, New York 10018
Title:
American National Standard for Information Systems -
Programming Language - C
ANSI X3.159-1989
```

Library Conventions

All function names and names of global variables are lowercase symbols.

The C Runtime Library is a small model library, that means there is only one data segment.

Currently, the C Runtime Library is for the most part non-reentrant. A CP/Q task that creates another task from itself, and both tasks use the same C Runtime Library executable code in the same address space encounter unexpected results due to the non-reentrant nature of the library. The C Runtime Library is reentrant on a per load module basis. Two separate C programs loaded into the same address space, each having a single task executing the program, is completely reentrant because each tasks is executing within a different load module, using different copies of C Runtime Library static data. The problem occurs when two or more threads of execution (tasks) are executing within the same physical code, using the same copy of C Runtime Library static data. The C Runtime Library contains static data areas that is potentially corrupted when two or more tasks are concurrently modifying this static data.

The main() function

On the call to *main()* there exist two parameters, the first parameter is the number of arguments and the second parameter is a pointer to the list of arguments. Each element of the argument list is a pointer to a null terminated string. The argument pointed to by *argv[argc]* is NULL and indicates the end of the command line arguments. The C *main()* function prototype is defined as follows by the CP/Q C Runtime Library implementation:

```
main(argc, argv)
 int   argc;
 char *argv[];
```

When a program returns from *main()*, a return value is generated either by a return instruction within *main()* or by falling off the end of the *main()* function. In either case, the return value from *main()* is presented to the program's fault handler when

the C program terminates. A zero return code is considered a successful termination by convention. A non-zero return code indicates an unsuccessful return code. For more information on fault handlers, see *Systems Programming Library Volume 3: SVC Handler*.

Startup, Initialization and Exit

Startup is the entry point for C programs that are being created to run under CP/Q. The C Runtime Library module, clib, contains C startup and initialization code that performs the necessary actions to setup stdio buffers and command line argument parsing for *argv[argc]*, among other things. The C programmer can choose between three different pairs of C initialization/exit code, based upon the program's requirements and size:

cinit, exit

The default issued C initialization and exit code provided in the C Runtime Library archive module, *clib*. This is what most C programs use if no other set of C initialization/exit code is specified when linking a program. The advantage with this set of initialization/exit code is that there are no restrictions in calling other C functions. The disadvantage is that the resultant load module is larger because cinit and exit include all the C Runtime Library file I/O and buffer routines as well as some time functions within the resultant load module.

The C initialization code within cinit parses the command line, opens the stdio streams, calls the *clock()* function to establish a starting time for the program, and then calls *main()*. The C exit code contained within exit is called when the *main()* function returns. This version of the C exit code calls the *atexit()* functions, flushes and closes all open streams, deallocates all memory objects created by memory allocation functions (ie: *malloc()*, *realloc()*, *calloc()*, etc...), deallocates all memory object created by environment variable functions (ie: *putenv()*), and then halts the program by calling **CPHaltTask** with the return code returned from the *main()* function.

cinitnon, exitnon

This version of C Runtime Library initialization/exit code provides for the smallest resultant load module. The disadvantage is that there are restrictions on its use. The C program cannot perform stream I/O and cannot read from or write to the stdio streams stdin, stdout, and stderr because the stream I/O code is omitted from this version of the C initialization code. The *clock()* function is not called either as in other versions of C initialization code. All that is performed in this version of C initialization code is that the command line arguments are parsed and *main()* is called. The C exitnon code is called when *main()* returns. exitnon calls the *atexit()* functions, deallocates all memory objects created by memory allocation functions (ie: *malloc()*, *realloc()*, *calloc()*, etc...), deallocates all memory object created by environment variable functions (ie: *putenv()*), and then halts the program by calling **CPHaltTask** with the return code returned from the *main()* function.

See “Compiling and Linking a Program” on page 1-4 for instructions on how include different versions of C initialization/exit code.

Heap Management

SLEEP/R and the CP/Q Loader can create an initial heap for use by the loaded C program. The size of the heap is increased if necessary by requesting additional address space from the Memory Manager. The additional heap space added can be discontiguous with the initial heap. This feature is taken into account by the heap management routines. For a full description of the heap and its structure, see "Structure of the Heap" on page 19-5.

Environment Management

SLEEP/R and the CP/Q Loader create an environment data area for the loaded C program. The environment data area contains pointers to environment variable data that were copied from the creator task. These environment variables can be read or modified by the loaded C program. Currently, there is no provision for sharing environment variables between tasks. The environment data area also contains fields to be used by a C program for heap management and stdio stream(s). For a full description of the of the environment data area and its structure, see "Environment Variable Implementation" on page 19-8.

The address space in use by environment variables can be increased if necessary by requesting additional address space from the Memory Manager in the same way the heap management routines request additional address space. The additional environment variable space added can be discontiguous with the initial environment variable definitions copied from the creator task.

Screen and Keyboard

The Session Manager is used to access the keyboard and the screen. The first time something is actually written to stdout or stderr (which can not be until a buffer is flushed), a stdio session is created. The returned session ID is saved for any later requests to the keyboard or screen. For communication with the Session Manager, its library functions are used (SMLIB). These Session Manager functions in turn use CPQLIB system library functions for sending messages from the user program to the Session Manager task. To gain direct access to your session you can use the C Runtime Library *getsessid()* function, which returns the session ID used to access screen and keyboard for the stdio session. The *setsessid()* function can be used to change the stdio session. Before using *setsessid()* it is recommended that the stdin, stdout, and stderr streams are flushed by calling the C Runtime Library *fclose()* function.

User-defined copyright notices

The C Runtime Library provides a mechanism whereby users can place their own copyright notices near the beginning of a created load module.

CP/Q Version 1: Place the following assembler code example in a separate assembler source module, and insert the resultant XCOFF module BEFORE the C Runtime Library archive clib.lib in the link control file in order for the user defined copyright to appear on the load module code section near the beginning of the load module.

```
;*****************************************************************/
;* User defined copyright                                        */
;*****************************************************************/
      .386p
      TITLE    copyright
      NAME     copyright

code32      group   code
code        segment use32
            assume  cs:code32,ds:nothing,es:nothing,ss:nothing
;*****************************************************************/
;* copyright is required in the code section                     */
;*****************************************************************/
            public  _copyright

_copyright  db 'Place your copyright notice in here'

code        ends
            end
```

Patch Area

A 64 byte patch area is defined in the code segment of a C program that uses the C startup code provided in the C Runtime Library. The intended use of this patch area is for dynamically modifing a program under test using the Task Level Debugger (TLD). The TLD has facilities that allow a program under test to be modified or "patched".

The public symbol indicating the beginning of the patch area is called "_patcharea" (excluding quotes). The patch area initially contains 8 consecutive occurances of the string "PATCH***" (excluding quotes) so that the patch area is easy to locate.

Information on the TLD can be found in *Application Programming Library Volume 3: Programming Tools*.

Compiling and Linking a Program

INTEL Version (CP/Q Version 1): The following C compilers are supported as development tools to create CP/Q Version 1 programs:

- Metaware HighC Version 3 Optimizing C compiler (runs on DOS and in an OS/2 V2.x DOS box)
- IBM C/C++ Set/2 Compiler (runs on OS/2 V2.x)

Contact Metaware Incorporated, 2161 Delaware Avenue, Santa Cruz, CA 95060-5706 (Tel: (408) 429-6382 Fax: (408)429-9273) for ordering information concerning the Metaware HighC compiler for extended DOS. Contact IBM for ordering information concerning the IBM C/C++ Set/2 compiler.

The following assemblers are supported as development tools to create CP/Q Version 1 programs:

- Microsoft MASM 5.1 and Microsoft MASM 6.0

Below are two sample C source modules, *hello1.c* and *hello2.c*. These two modules are used to illustrate how to compile and link a program for CP/Q Version 1.

```
/*****************************************/
/* hello1.c module                       */
/*****************************************/
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

extern int printhello(void);


int main(int argc, char **argv)
{
int rc;

rc = printhello();

return(rc);
}
/*****************************************/
/* end of hello1.c module                */
/*****************************************/
```

Figure 1-1. HELLO1.C sample source module

```
/*****************************************/
/* hello2.c module                       */
/*****************************************/
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>


int printhello(void)
{
printf("Hello there\n");
return(EXIT_SUCCESS);
}
/*****************************************/
/* end of hello2.c module                */
/*****************************************/
```

Figure 1-2. HELLO2.C sample source module

Figure 1-3 on page 1-6 illustrates a UNIX-style make file that uses the Metaware HighC compiler to compile the hello program. Compiler switches for CP/Q development using Metaware HighC Version 3 are set by including the header file *highc.h*. *highc.h* gets automatically included into compiled source code by including any CP/Q ANSI C header file (ie: *stdio.h*,*stdlib.h*, *stddef.h*, etc).

Compiler options for Metaware HighC Version 3 can be set in a file called *hc.pro* within the current directory where the CP/Q source code is being compiled. A sample *hc.pro* file follows (all of the options in the sample *hc.pro* get set in *highc.h* as well):

```
#define I486
#define FORI486
pragma On(Read_only_strings);
pragma Ipath("c:\cpq\include;");
pragma Off(Align_members);
pragma Off(Use_BSS);
```

Figure 1-4 on page 1-7 illustrates a NMAKE-style make file that uses the IBM C/C++ Set/2 compiler to compile the hello program. The OS/2 Toolkit utility

```
#########################################################################
#                                                                       #
# Makefile to create the HELLO.XLD program using the Metaware HighC     #
# Version 3 compiler and the BIND program.                              #
#                                                                       #
# Notes:                                                                #
#                                                                       #
#  This makefile can be run on a DOS based system that has a UNIX-style #
#  make program.                                                        #
#########################################################################
CPQROOT = \cpq

#
# Directory macros
#
CPQ_D   = $(CPQROOT)
INC_D   = $(CPQ_D)/include
BIN_D   = $(CPQ_D)/progs
LIB_D   = $(CPQ_D)/lib

#
# The Librarian
#
AR      = ar

#
# The Linker
#
LD      = bind

#
# The Compiler
#
CC      = command /C hc386
CC_INC  = -c -I$(CPQROOT)\include
CFLAGS  = $(CC_INC)

#
# A few new suffixes
#
FFIXES:      .asm .c .xcf

#
# Inference Rules
#
.asm.xcf:
                masm /mx /z  $*,$*,$*,NUL
                @ cvox  $*.obj
                @ rm -f $*.obj

.c.xcf:
                $(CC) $*.c $(CFLAGS)
                @ cvox  $*.obj
                @ rm -f $*.obj

all:  hello.xld

hello.xld: hello1.xcf hello2.xcf
          $(LD) < $V$*.lnk
```

Figure 1-3. HELLO.MAK makefile using HighC as the compiler

NMAKE.EXE can be used to make to process either makefile. The makefile in Figure 1-3 on page 1-6 can be processed by a UNIX-style make program running under DOS. The makefile in Figure 1-4 on page 1-7 can be processed by the NMAKE.EXE OS/2 Toolkit utility under OS/2 V2.x.

After successful compilation, a utility called *CVOX* is run against the resulting object file to convert it to XCOFF format. *CVOX* is provided as part of the standard CP/Q distribution in three forms: one that runs under DOS (CVOX.EXE), another that runs under CP/Q (CVOX.XLD), and a third that runs under OS/2 V2.x (CVOX2.EXE).

CVOX requires one argument, the object module (.obj) produced by the Metaware HighC compiler. After successfully running *CVOX* against an object module, the resulting output is an XCOFF version of the object module that has a file extension indicating it is an XCOFF module (.xcf). The C Runtime Library is made up of a number of XCOFF files.

```
#########################################################################
#                                                                       #
# Makefile to create the HELLO.XLD program using IBM C/C++ Set/2        #
# as the compiler.                                                      #
#                                                                       #
# Notes:                                                                #
#                                                                       #
#  This makefile can be invoked by the OS/2 Toolkit program NMAKE.EXE   #
#  as follows:                                                          #
#                                                                       #
#                nmake -f hello.mak                                     #
#                                                                       #
#  The following compiler switches are used to create CP/Q programs     #
#                                                                       #
#         /Gs+   Remove stack probes from code.                         #
#         /Gx+   (C++ only) remove C++ exception handling information.  #
#         /Ms    Use _System linkage for functions.                     #
#         /Rn    Generate executable code that can be used as a         #
#                subsystem without a runtime environment.               #
#         /Sa    Conform to ANSI C standards.                           #
#         /Sc    (C++ only) allow older C++ constructs.                 #
#         /Sp1   Align structures along 1 byte boundaries.              #
#         /Ti-   Do not generate debugger information - use this        #
#                when optimization is turned on (/O+).                  #
#                Optimization is turned off by default.                 #
#         /C     Compiler source module only, no link.                  #
#         /Xi+   Do not search paths defined by the INCLUDE             #
#                environment variable.                                  #
```

Figure 1-4 (Part 1 of 3). HELLO.MAK makefile using IBM C/C++ Set/2 as the compiler

```
#                                                                       #
#  The following compiler option is required for CP/Q programs          #
#  that run as a supervisor task (PL0).  It also insures          #
#  a full stack frame is always present for debugging purposes:         #
#                                                                       #
#         /Op-   Do not perform optimizations that involve the stack    #
#                pointer.                                               #
#                                                                       #
#  The following option can be used to optimize code:                   #
#                                                                       #
#         /O+    Optimize code.  By default, no optimizations are       #
#                performed.  When optimization is on, the IBM C/C++     #
#                Set/2 compiler by default generates code instead of    #
#                a function call for the following C library functions: #
#                                                                       #
#                abs         memmove   strncat                          #
#                _clear87    memset    strncmp                          #
#                _control87  _status87 strncpy                          #
#                fabs        strcat    strrchr                          #
#                labs        strchr                                     #
#                memchr      strcmp                                     #
#                memcmp      strcpy                                     #
#                memcpy      strlen                                     #
#                                                                       #
#                When you #include the appropriate header file in which #
#                the function prototype are found, the compiler         #
#                generates code instead of a function call for these    #
#                functions.                                             #
#                                                                       #
#  C Set++ at "fix level 6" (CSD6) and above can support the following  #
#  option:                                                              #
#                                                                       #
#         /Yp+   Do not generate code which places the argument count   #
#                in register AL.  This is for "system" linkage only.    #
#                The option is used if you have the proper       #
#                version of the compiler.                               #
#                                                                       #
#  Optional compiler switches include:                                  #
#                                                                       #
#         /Fa+    Generate an assembler listing.                        #
#         /Ls+    Produce a listing file which includes the source      #
#                 code.                                                 #
#         /Oi     Controls inlining of USER functions.                  #
#         /Q      Supress compiler logo output.                         #
#                                                                       #
#########################################################################
CPQROOT = \cpq

#
# Directory macros
#
CPQ_D   = $(CPQROOT)
INC_D   = $(CPQ_D)/include
BIN_D   = $(CPQ_D)/progs
LIB_D   = $(CPQ_D)/lib
```

Figure 1-4 (Part 2 of 3). HELLO.MAK makefile using IBM C/C++ Set/2 as the compiler

```
#
# The Librarian
#
AR      = ar2

#
# The Linker
#
LD      = bind2

#
# The Compiler
#
CC      = icc
CC_INC  = /Xi /I$(CPQROOT)\include;
# Use the next line for ship level code
CC_OPT  = /Gs+ /Ms /Rn /Sa /Sp /O+ /Op-
# Use the next line for debugging code
# CC_OPT  = /Gs+ /Ms /Rn /Sa /Sp /Ti /Op-

CFLAGS  = $(CC_OPT) $(CC_DEF) $(CC_INC)


#
# A few new suffixes
#
FFIXES:      .asm .c .xcf

#
# Inference Rules
#
.asm.xcf:
                masm /mx /z  $*,$*,$*,NUL
                @ cvox2  $*.obj
                @ rm -f $*.obj

.c.xcf:
                $(CC) /C $(CFLAGS) $*.c
                @ cvox2  $*.obj
                @ rm -f $*.obj

all:  hello.xld

hello.xld: hello1.xcf hello2.xcf
           $(LD) < $V$*.lnk
```

Figure 1-4 (Part 3 of 3). HELLO.MAK makefile using IBM C/C++ Set/2 as the compiler

In order to link or "bind" the program into an executable entity that can be run under CP/Q, run a program called *BIND BIND* is provided as part of the standard CP/Q distribution in three forms: one that runs under DOS (BIND.EXE), another that runs under CP/Q (BIND.XLD), and a third that runs under OS/2 V2.x (BIND2.EXE).

BIND requires a link control file to produce a CP/Q executable (.xld) file. Figure 1-5 on page 1-10 illustrates the link control file for producing HELLO.XLD. The sample link control file can be used by either makefile shown above.

```
* This is a sample link command file.  Comments require a * in
* column one.  You can redirect standard input from this file,
* or you can start the binder and then use the EXECute command
* to run this file.

setopt quiet
setopt wantkey
setopt verbose

* Place your object files here:
insert hello1.xcf
insert hello2.xcf

insert \cpq\lib\clib.lib
insert \cpq\lib\cpqlib.lib
insert \cpq\lib\conlib.lib
insert \cpq\lib\fslib.lib
insert \cpq\lib\ldlib.lib
insert \cpq\lib\smlib.lib
insert \cpq\lib\unixlib.lib

* Resolve external references between object files:
resolve

* Main load module entry point (note that symbol names are
* case sensitive):
entry startup

* Garbage Collect unused routines so they are not included in the
* load module:
gc

* Check for unresolved external references:
er full

* Save the load module to a file (our convention is that
* executable XCOFF files use the extension .XLD to distinguish
* them from non-executables):
save 1l hello.xld
```

Figure 1-5. HELLO.LNK link control file

A sample link control file is provided in the standard CP/Q distribution and is called SAMPLE.LNK. A sample link control file is also listed in Appendix C, “Sample Link Control File” on page C-1. If a different version of C initialization/exit code is to be included to override the default version within the library, place the object files containing the C initialization/exit code before the 'insert' statement for *clib.lib* within the link control file. C initialization/exit object code modules are provided with the C Runtime Library distribution package. See “Startup, Initialization and Exit” on page 1-2 for an explanation on different versions of C initialization/exit code. More information concerning the *BIND* program can be found in the LDBIND LIST3820 document provided as part of the standard CP/Q distribution.

Floating Point Support

The C Runtime Library does provide limited floating point support for systems that contain a numeric coprocessor. The *printf()* and *scanf()* family of functions correctly handle floating point values and the *atof()* function is provided to convert from string to floating point types. However, other floating point functions declared in the ANSI C standard are missing and are added as time allows.

Object Module Format

The object module format for CP/Q is XCOFF (COFF Extended Object File Format). For more information on the XCOFF format, see the XCFOBJ LIST3820 document provided as part of the standard CP/Q distribution.

Library Creation

To create your own XCOFF module libraries, a program called *AR* is provided as part of the standard CP/Q distribution in three forms: one that runs under DOS (AR.EXE), another that runs under CP/Q (AR.XLD), and a third that runs under OS/2 V2.x (AR2.EXE). For more information on *AR*, see *Systems Programming Library Volume 9: XCOFF Utilites*. The AR program is used to create the C Runtime Library.

Chapter 2. Global Variables

The intent of this chapter is to list the global variables defined by the C Runtime Library.

Any use or redefinition of the following global variables than what are described within this chapter is not supported in the CP/Q C Runtime Library design and is HIGHLY discouraged.

Some global variables are defined in the module cjldata. Other global variables are defined in different modules so that extraneous modules do not get 'pulled in' during link time.

The C Runtime Library global variables are intended to emulate the ANSI C standard as closely as possible. However, some extensions have been added to support CP/Q operating system functions.

_cdapage

```
Classification -

     global variable

Declaration -

     CDA  *_cdapage

Location -

     sysdep.h
```

Description

Points to the user-mode read only CDA page. The structure *CDA* will be contained in the header file cda.h. This offset can be used to get the time, date, or the release and version numbers of the kernel components for example.

Restrictions

This pointer is not to be altered by a program.

Implementation Notes

The global variable *_cdapage* is initialized in the C startup code. The startup routine in CSTART will call the SVC to obtain the user-mode read only CDA page. If the call to obtain the offset of the user mode CDA page fails, the program will abort at this point.

If the module CSTART is not used, then it is up to the task to initialize this global variable.

_cjlvsn

```
Classification -

     global variable

Declaration -

     const char *cjlvsn

Location -

     sysdep.h
```

Description

Points to the version string of the C Runtime Library. The version string is 12 bytes long (including NULL) and has the following format:

CP/Q Vx Dyy

Where "x" is the version number and "yy" is the release number of the C Runtime Library.

Restrictions

This pointer is not to be altered by a program.

errno

```
Classification -

     global variable

Declaration -

     int errno

Location -

     errno.h
```

Description

The global variable *errno* contains the error status of library functions that wish to return information about any error that occurs during its execution. A number other than zero contained within *errno* indicates an error.

It is highly recommended that the external definition for *errno* be obtained by including the file *errno.h*. *errno* is set to zero on program startup.

Restrictions

An error code can get overwritten on consecutive errors.

_fullname

```
Classification -

     global variable

Declaration -

     char *_fullname

Location -

     stdio.h
```

Description

Points to the fully qualified path and filename of the executing program. The filename defined here might be different from the argv[0] variable because argv[0] may contain an alias to the filename. The filename pointed to by *_fullname* is never an alias.

Restrictions

This pointer is not to be altered by a program.

_mons

```
Classification -

     global variable

Declaration -

     const char *_mons[]

Location -

     time.h
```

Description

Defined as an array of 3-byte strings that contain abbreviations for the months of the year. This array is used by the *asctime* and *strftime* functions.

Restrictions

This array is not to be altered by a program.

_wdays

```
Classification -

     global variable

Declaration -

     const char *_wdays[]


Location -

     time.h
```

Description

Defined as an array of 3-byte strings that contain abbreviations for the days of the week. This array is used by the *asctime* and *strftime* functions.

Restrictions

This array is not to be altered by a program.

Chapter 3. Allocation Functions

calloc

```
Declaration -

     #include "stdlib.h"

     void *calloc(no, size)
         size_t no;
         size_t size;

Return Codes -

     NULL    -  insufficient memory available.
     ptr     -  pointer to the allocated space.
```

Usage Notes:

This function allocates contiguous heap space of *no* times *size* bytes. The allocated storage is initialized to zero.

free

```
Declaration -

     #include "stdlib.h"

     void free(ptr)
         void *ptr;

Return Codes -

     None
```

Usage Notes:

This function frees storage allocated via *malloc*, *calloc* or *realloc*.

malloc

```
Declaration -

     #include "stdlib.h"

     void *malloc(size)
         size_t size;

Return Codes -

     NULL    -  insufficient memory available.
     ptr     -  pointer to the allocated space.
```

Usage Notes:

This function allocates contiguous heap space of *size* bytes. The allocated storage is not initialized.

Implementation Notes:

Additional address space is requested from the Memory Manager if more heap space is needed. It is possible that the additional heap space will not be contiguous with existing heap space.

Heap objects are created as either user or supervisor mode, depending on the currently executing privilege level and are owned by the caller's process.

realloc

```
Declaration -

     #include "stdlib.h"

     void *realloc(oldptr, size)
         void *oldptr;
         size_t size;

Return Codes -

     NULL    -  insufficient memory available or oldptr invalid.
     ptr     -  pointer to the allocated space.
```

Usage Notes:

This function allocates contiguous heap space of *size* bytes and move the contents from the location given by *oldptr* to the new location.

Implementation Notes:

Additional address space is requested from the Memory Manager if more heap space is needed. It is possible that the additional heap space will not be contiguous with existing heap space.

Heap objects are created as either user or supervisor mode, depending on the currently executing privilege level and are owned by the caller's process.

Chapter 4. Environment Functions

copyenv

```
Declaration -

     #include "stdlib.h"

     int copyenv(totaskSVid, envstart, envfree, envobj, mem_ptr, fromtaskSVid, mode);
         UINT32 totaskSVid;
         UINT32 *envstart;
         UINT32 *envfree;
         UINT32 *envobj;
         UINT32 *mem_ptr;
         UINT32 fromtaskSVid;
         int   mode;

Return Codes -

     nnnn    -  environment variables not copied.
     NULL    -  environment variables successfully copied.
```

Usage Notes:

This special purpose function is intended to copy environment variables from one task to another. The only application intended to use this function (right now) is the LOADER. *copyenv()* requires set-effective-process privilege and it is highly recommended that a task using *copyenv()* be running at PL0 in order to make aliases to PL0 data objects.

copyenv() requires the source task SVid to be placed in the parameter *fromtaskSVid* and the destination task SVid to be placed in *totaskSVid*. The *mode* parameter defines the privilege of the data objects created for use by environment variables in the address space of *totaskSVid*. The *mem_ptr* parameter may be NULL or passed the address of memory already allocated by the program calling *copyenv()*. (In the case of the LOADER, the initial heap space can be used.) If *mem_ptr* is NULL, a new memory page will be allocated for environment variables. If this parameter is non-NULL, the *copyenv* function will copy environment variables into the initial memory space pointed to by *mem_ptr* instead of allocating additional memory pages for environment variable storage. The memory space pointed to by *mem_ptr* does not to be initialized in any way before calling *copyenv()*. The *mode* parameter may be either *QMuser* or *QMsupervisor*.

Upon return from the *copyenv()* function, the parameters *envstart*, *envfree*, and *envobj* will contain values to be put in the environment data area (see the section "Environment Variables Implementation" contained in the Technical Reference chapter for an explanation of the environment data area). If *envstart* contains a NULL value upon return from the *copyenv()* function, then it can be assumed that *envfree* and *envobj* are both NULL, no environment data area needs to be created

and the environment data area pointer is NULL (the environment data area pointer field is defined in the "C Startup Register Interface" section contained in the Technical Reference chapter). Also upon return, the parameter *mem_ptr* will contain the address of a pointer to an area of memory that was unused in the purpose of copying environment varaibles if a non-NULL value was passed in to *mem_ptr*. The *copyenv()* function will use the memory pointed to by *mem_ptr* for the purpose of copying environment variables instead of allocating another memory page. The *copyenv()* function will use this previously allocated memory to store environment variables, and then return a value in *mem_ptr* to the remaining memory space originally given to it that is not used. This returned memory space is formatted as a heap object. If the environment area being copied overflows the memory area pointed to by *mem_ptr*, then *copyenv()* will allocate more memory for environment storage and return *mem_ptr* equal to NULL. The idea here is that returning a pointer to a memory object different than what was passed to you may not be a good idea.

A task that has no environment data area has not been passed any environment variables, but still can create environment variables via the *putenv()* function. These created environment variables will be copied to another task when this task calls the LOADER to load a load module.

If anything goes wrong within the *copyenv()* function, such as a bad return code from a call to *CPAlias()* or *CPAllocMem*, the *copyenv()* function will try to clean up any aliases it has created in the "source"(from) and "destination" (to) address spaces and reset the effective process of the task calling the *copyenv()* function to what the effective process was before entering this function. The last return code received by the *copyenv()* function will be returned to the caller.

Implementation Notes:

This function will "chase" down the linked list of environment objects in the address space of *fromtaskSVid* and "copy" these environment objects into the address space of *totaskSVid* while adjusting the forward pointer values contained the environment objects that were copied. Aliases are extensively used by *copyenv()* to perform its function.

Environment variables are copied on a "per-task" basis. There is currently no provision for sharing environment variables between tasks - environment areas are copied from the task calling the CP/Q Loader to the task containing the newly loaded load module.

getenv

```
Declaration -

     #include "stdlib.h"

     char *getenv(envvar)
         const char *envvar;

Return Codes -

     ptr     -  string returned that defines the environment variable.
     NULL    -  could not find the environment variable.
```

Usage Notes:

This function returns the string that defines the environment variable *envvar*. If *envvar* is not currently defined in the environment area, then a NULL value is returned.

Note that only the environment variable definition is returned. The environment variable name and the equal ('=') symbol are not contained in the string returned by this call.

Implementation Notes:

Environment variables are copied on a "per-task" basis. There is currently no provision for sharing environment variables between tasks - environment areas are copied from the task calling the CP/Q Loader to the task containing the newly loaded load module.

getenvall

```
Declaration -

     #include "stdlib.h"

     int getenvall(envbufptr)
         char **envbufptr;

Return Codes -

     NULL    -  successful completion.
     EOF     -  could not allocate buffer or bad parameter value.
```

Usage Notes:

This function will dynamically allocate a buffer and place all environment variables and their definitions for the current C program in the allocated buffer. The parameter *envbufptr* will point to this allocated buffer upon successful completion of the *getenvall()* call.

An advantage of using *getenvall* over *getenvall2* is that *getenvall* will try to create a buffer large enough to contain all environment variables for the current C program. Whereas *getenvall2* can be passed a buffer that is not large enough to contain all environment variable and their definitions.

Implementation Notes:

The format of this allocated buffer is as follows:

environment variable = definition <NULL>
environment variable = definition <NULL>
environment variable = definition <NULL>
environment variable = definition <NULL>
environment variable = definition <NULL>
. .
. .
. .
environment variable = definition <NULL>
<NULL>

That is, a string containing the environment variable and its defintion with a terminating NULL character for each environment variable defined within a task. The end of the buffer is signalled by two consecutive NULL characters. If there are no environment variables defined within a task, a call to *getenvall()* will yield a buffer of two NULL characters.

This function uses heap space to dynamically allocate the buffer.

getenvall2

```
Declaration -

     #include "stdlib.h"

     int getenvall2(envbufptr, envbuflen)
         char *envbufptr;
         int   envbuflen;

Return Codes -

     NULL    -  successful completion.
     EOF     -  invalid argument(s) or buffer filled before
                copy procedure complete
```

Usage Notes:

This function will copy all the environment variables and their definitions for the current C program into the buffer supplied by the caller. *envbufptr* points to the buffer to be filled with environment variables and their definitions. *envbuflen* specifies the length of the buffer in bytes. If the buffer pointed to by *envbufptr* is filled before all environment variable information is copied into the buffer, EOF will be returned. A return value of EOF will also be returned if either argument is NULL.

getenvall2 differs from *getenvall* in that *getenvall* will create the buffer, whereas *getenvall2* is supplied a buffer by the caller.

Implementation Notes:

Upon successful completion, the format of the environment variable information contained in the supplied buffer is as follows:

environment variable = definition <NULL>
environment variable = definition <NULL>
environment variable = definition <NULL>
environment variable = definition <NULL>
environment variable = definition <NULL>
. .
. .
. .
environment variable = definition <NULL>
<NULL>

That is, a string containing the environment variable and its defintion with a terminating NULL character for each environment variable defined within a task. The end of the buffer is signalled by two consecutive NULL characters. If there are no environment variables defined within a task, a call to *getenvall2()* will yield a buffer of two NULL characters.

This function does not use any heap space.

putenv

```
Declaration -

     #include "stdlib.h"

     int putenv(envstring)
         char *envstring;

Return Codes -

     EOF     -  failure in adding/replacing an environment variable value.
     NULL    -  successful completion.
```

Usage Notes:

This function will add, replace or clear an environment variable value. The character string pointed to by *envstring* must have the following format:

environment variable[=*value*]

Where *environment variable* represents the environment variable name and *value* represents the environment variable definition. Note the equal sign ('=') between *environment variable* and *value* - the equal sign is required in the character string when specifying *value*.

An environment variable and its definition are added when *putenv* is called with an environment variable that is not currently defined in the environment variable area. The environment variable and its definition are added to the environment variable area.

An environment variable definition is replaced when *putenv* is called with a environment variable that is currently defined in the environment variable area. The previous environment variable and its definition are replaced in the environment variable area with the environment variable and the defintion specified in the *putenv* call.

An environment variable is cleared (or removed) when the *putenv* call specifies a parameter string that only contains *environment variable*. That is, *value* is not present in the passed parameter string. In this case, *environment variable* and its value are removed from the environment variable area.

Implementation Notes:

Case insensitive searches are performed in the environment variable area. This means "PATH" and "path" refer to the same environment variable.

The maximum length of *envstring* can be 4096-8 = 4088 bytes.

Environment variables are copied on a "per-task" basis. There is currently no provision for sharing environment variables between tasks - environment areas are copied from the task calling the CP/Q Loader to the task containing the newly loaded load module.

Chapter 5. File Functions

clearerr

```
Declaration -

     #include "stdio.h"

     void clearerr(stream)
         FILE *stream;

Return Codes -

     NONE
```

Usage Notes:
This macro clears error and end-of-file indicators.

Implementation Notes:
None.

fclose

```
Declaration -

     #include "stdio.h"

     int fclose(stream)
         FILE *stream;

Return Codes -

     0       -  stream successfully closed.
     EOF     -  error occurred while closing stream.
```

Usage Notes:

The associated buffer is flushed before closing. If the buffer was allocated by the system, that means not assigned via a *setbuf* or *setvbuf* call, it will be freed.

Implementation Notes:

None.

feof

```
Declaration -

     #include "stdio.h"

     int feof(stream)
         FILE *stream;

Return Codes -

     == 0    -  end-of-file has not been reached.
     != 0    -  end-of-file has been reached.
```

Usage Notes:

This macro determines whether the end of *stream* has been reached.

Implementation Notes:

None.

ferror

```
Declaration -

     #include "stdio.h"

     int ferror(stream)
         FILE *stream;

Return Codes -

     == 0    -  no error has occurred.
     != 0    -  an error has occurred.
```

Usage Notes:

This macro determines whether an error on *stream* has occurred.

Implementation Notes:

None.

fflush

```
Declaration -

     #include "stdio.h"

     int fflush(stream)
         FILE *stream;

Return Codes -

     EOF     -  error encountered.
     0       -  buffer successfully flushed.
```

Usage Notes:

This function causes the buffer associated with *stream* to be written to the file. If the buffer is in read status nothing happens and a NULL is returned.

Implementation Notes:

None.

fgetc

```
Declaration -

     #include "stdio.h"

     int fgetc(stream)
         FILE *stream;

Return Codes -

     EOF     -  error or end-of-file encountered.
     c       -  ASCII value of character read.
```

Usage Notes:

This function returns the next character from *stream*.

Implementation Notes:

None.

fgetpos

```
Declaration -

     #include "stdio.h"

     int fgetpos(stream, offptr)
         FILE *stream;
         fpos_t *offptr;

Return Codes -

     != 0    -  error occurred, offptr undetermined and errno set.
     == 0    -  offptr contains current file position.
```

Usage Notes:

This function stores the current position of the file pointer relative to the beginning of the file associated with *stream* in the location given by *offptr*.

Implementation Notes:

None.

fgets

```
Declaration -

     #include "stdio.h"

     char *fgets(string, n, stream)
         char *string;
         integer n;
         FILE *stream;

Return Codes -

     NULL    -  error found or only end-of-file read.
     ptr     -  equals string.
```

Usage Notes:

This function reads from *stream* until *n*-1 characters are read, a newline character is read or end-of-file is seen. If the last character read was a newline, the newline character will be included in *string*. The *string* parameter is terminated with a null character.

Implementation Notes:

None.

fileno

```
Declaration -

     #include "stdio.h"

     int fileno(FILE *stream)
         FILE *stream;

Return Codes -

     nn      -  returned file descriptor
```

Usage Notes:

This macro returns the integer file descriptor associated with *stream*.

Implementation Notes:

The File System file descriptor is returned.

fopen

```
Declaration -

     #include "stdio.h"

     FILE *fopen(filename, mode)
         const char *filename;
         const char *mode;

Return Codes -

     NULL    -  file not opened.
     fptr    -  pointer to a file control block.
```

Usage Notes:

This function opens the file specified by *filename*.

The *mode* parameter specifies the type of access requested for the stream.

Table 5-1. File access types

Mode	Meaning
"r"	Open a file for reading. Failure if the file doesn't exist.
"w"	Open file for writing. If the file exists, its contents are destroyed.
"a"	Open file for writing at the end of the file. If the file doesn't exist it will be created.
"r+"	Open file for reading and writing. Failure if the file doesn't exist.
"w+"	Open file for reading and writing. If the file already exists. its contents are destroyed.
"a+"	Open for reading and writing at the end of the file. If the file doesn't exist it will be created.

If the stream is opened for binary access, this implies that no translation of 0x0D and 0x0A combinations take place. This also implies that the CONTROL-Z (0x1A) is not treated as a end of file character.

Appending a 'b' to the *mode* parameter opens the file in binary mode.

Appending a 't' to the *mode* parameter opens the file in text mode. All the actions excluded for the binary mode are now performed. This is the default file mode.

If you open a file in update mode use *fseek* or *rewind* to clear internal buffers.

Implementation Notes:

If the *mode* is specified as "append" or "append/update" in text mode, then *fopen()* will automatically check for a EOF character (0x1A) at the end of the opened file. If the EOF character is present, the file pointer will be decremented so as to "back up" over the EOF character before any data is written to the stream. This avoids writing data after the EOF character in text mode, which would have the effect of the data appearing never to have been written when the data is read back because a stream in text mode ends when the first EOF character (0x1A) is seen.

fputc

```
Declaration -

     #include "stdio.h"

     int fputc(c, stream)
         int   c;
         FILE *stream;

Return Codes -

     EOF     -  error encountered.
     c       -  character written.
```

Usage Notes:

This function writes one character to *stream*.

Implementation Notes:

None.

fputs

```
Declaration -

     #include "stdio.h"

     int fputs(string, stream)
         const char *string;
         FILE *stream;

Return Codes -

     EOF     -  error encountered.
     nnn     -  string written to stream (non-negative number returned).
```

Usage Notes:

This function copies *string* to *stream* up to the terminating null character. No newline character is appended. *fputs()* returns EOF if an error occurs; otherwise, it returns a non-negative number.

Implementation Notes:

None.

fread

```
Declaration -

     #include "stdio.h"

     size_t fread(ptr, ptrsize, nitems, stream)
         void *ptr;
         size_t ptrsize;
         size_t nitems;
         FILE *stream;

Return Codes -

     nn      -  number of items read.
```

Usage Notes:

This function reads from *stream ptrsize* times *nitems* bytes into the location given by *ptr*. The number of completely read items is returned. A return value less than *nitems* usually indicates an error condition.

Implementation Notes:

None.

freopen

```
Declaration -

     #include "stdio.h"

     FILE *freopen(filename, mode, stream)
         const char *filename;
         const char *mode;
         FILE *stream;

Return Codes -

     NULL    -  file not opened.
     fptr    -  original value of stream.
```

Usage Notes:

This function closes the file currently associated with *stream* and then opens the file specified by *filename* using the slot just freed.

For the *mode* values and their meaning refer to the description of *fopen*.

Implementation Notes:

None.

fseek

```
Declaration -

     #include "stdio.h"

     int fseek(stream, offset, refpt)
         FILE *stream;
         long  offset;
         int   refpt;

Return Codes -

     == 0    -  file pointer successfully moved to new location.
     != 0    -  invalid seek request.
```

Usage Notes:

This function sets the file pointer to the requested position in *stream*. The *offset* is relative to the given *refpt* value.

Table 5-2. Seek reference points.

Numerical value	Symbolic name	Description
0	SEEK_SET	Beginning of file
1	SEEK_CUR	Current position
2	SEEK_END	End of file

You may use *fseek* to clear internal buffers. This is useful if you switching between read and write access on files opened for update.

Implementation Notes:

fseek will not return an error when a seek occurs past the end-of-file(eof). Instead, the file length is extended to incorporate the new file pointer position.

fsetpos

```
Declaration -

     #include "stdio.h"

     int fsetpos(stream, offptr)
         FILE *stream;
         const fpos_t *offptr;

Return Codes -

     nn      -  error occurred.
     0       -  file pointer successfully moved.
```

Usage Notes:

This function moves the file pointer of *stream* to the offset found in the location given by *offptr*. The offset is relative to the beginning of the file.

Implementation Notes:

None.

ftell

```
Declaration -

     #include "stdio.h"

     long int ftell(stream)
         FILE *stream;

Return Codes -

     -1L     -  error occurred and errno is set.
     nn      -  current offset.
```

Usage Notes:

This function returns the current position of the file pointer relative to the beginning of the file associated with *stream*.

Implementation Notes:

None.

fwrite

```
Declaration -

     #include "stdio.h"

     size_t fwrite(ptr, ptrsize, nitems, stream)
         const void *ptr;
         size_t ptrsize;
         size_t nitems;
         FILE *stream;

Return Codes -

     nn      -  number of items written.
```

Usage Notes:

This function copies *ptrsize* times *nitems* bytes from the location given by *ptr* to *stream*. The number of completely written items is returned. A return value less than *nitems* usually indicates an error condition.

Implementation Notes:

None.

getc

```
Declaration -

     #include "stdio.h"

     int getc(stream)
         FILE *stream;

Return Codes -

     EOF     -  error or end-of-file encountered.
     c       -  ASCII value of character read.
```

Usage Notes:

This macro returns the next character from *stream*.

Implementation Notes:

The *getc* macro evaluates its parameter more than once, leaving open the possibility of side-effects upon execution. This means that changing the value of the parameter in a call to *getc* can result in incorrect execution because the parameter may be changed to a value not intended by the programmer.

getchar

```
Declaration -

     #include "stdio.h"

     int getchar(void)

Return Codes -

     EOF     -  error or end-of-file encountered.
     c       -  ASCII value of character read.
```

Usage Notes:
This macro is identical to getc(*stdin*).

Implementation Notes:
None.

gets

```
Declaration -

     #include "stdio.h"

     char *gets(string)
         char *string;

Return Codes -

     NULL    -  error found or only end-of-file read.
     ptr     -  equals string.
```

Usage Notes:

This function reads from stdin until a newline character is seen. The *string* parameter is terminated with a null character. The newline character is replaced with the null character.

Implementation Notes:

None.

putc

```
Declaration -

     #include "stdio.h"

     int putc(c, stream)
         int   c;
         FILE *stream;

Return Codes -

     EOF     -  error encountered.
     c       -  character written.
```

Usage Notes:

This macro writes one character to *stream*.

Implementation Notes:

The *putc* macro evaluates its parameter more than once, leaving open the possibility of side-effects upon execution. This means that changing the value of the parameter in a call to *putc* can result in incorrect execution because the parameter may be changed to a value not intended by the programmer.

putchar

```
Declaration -

     #include "stdio.h"

     int putchar(c);
         int   c;

Return Codes -

     EOF     -  error encountered.
     c       -  character written.
```

Usage Notes:
This macro is identical to putc(c, stdout).

Implementation Notes:
None.

puts

```
Declaration -

     #include "stdio.h"

     int puts(string)
         const char *string;

Return Codes -

     EOF     -  error encountered.
     nnn     -  string written to stdout (non-negative value returned).
```

Usage Notes:

This function copies *string* to stdout up to the terminating null character. A newline character is appended. *puts()* returns EOF if an error occurs; otherwise, it returns a non-negative number.

Implementation Notes:

None.

remove

```
Declaration -

     #include "stdio.h"

     int remove(filename)
         const char *filename;

Return Codes -

     == 0    - file removed.
     != 0    - error in removing file.
```

Usage Notes:

This function removes *filename* so that a subsequent attempt to open the named file will fail.

Implementation Notes:

None.

rename

```
Declaration -

     #include "stdio.h"

     int rename(oldfilename,newfilename)
         const char *oldfilename;
         const char *newfilename;

Return Codes -

     == 0    - file renamed.
     != 0    - error in renaming file.
```

Usage Notes:
This function renames *oldfilename* to *newfilename*.

Implementation Notes:
None.

rewind

```
Declaration -

     #include "stdio.h"

     void rewind(stream)
         FILE *stream;

Return Codes -

     None
```

Usage Notes:

This macro moves the filepointer to the beginning of the file. The end-of-file indicator is cleared. If the error flag was set no action is taken.

Implementation Notes:

None.

setbuf

```
Declaration -

     #include "stdio.h"

     void setbuf(stream, string)
         FILE *stream;
         char *string;

Return Codes -

     NONE
```

Usage Notes:

This function controls buffering for the specified *stream*. If *string* equals NULL, the *stream* will be unbuffered. Otherwise *string* must be the location of a character array of length *BUFSIZ*.

This function must be used after the file is opened and before the first operation on a buffered *stream* has been done.

Implementation Notes:

None.

setvbuf

```
Declaration -

     #include "stdio.h"

     int setvbuf(stream, string, type, size)
         FILE *stream;
         char *string;
         int    type;
         size_t size;

Return Codes -

     == 0    -  buffer set appropriately.
     != 0    -  invalid setvbuf request.
```

Usage Notes:

This function controls buffering for the specified *stream*.

The *string* parameter is a character array of length *size* to be used as the buffer for *stream*.

Table 5-3. Setvbuf types

Type	Description
_IONBF	Unbuffered stream, the values for *string* and *size* must be zero.
_IOLBF	Line buffering, *string* and *size* are used to determine the size and the location of the buffer.
_IOFBF	Same as _IOLBF.

This function must be used after the file is opened and before the first operation on a buffered *stream* has been done.

Implementation Notes:

None.

tmpfile

```
Declaration -

     #include "stdio.h"

     FILE *tmpfile(void)

Return Codes -

     fptr    -   file pointer of temporary file created.
     NULL    -   file could not be created.
```

Usage Notes:

This function creates a temporary file of mode "w+b" that will be removed from the file system when the file is closed or when the program terminates normally.

Implementation Notes:

None.

tmpnam

```
Declaration -

     #include "stdio.h"

     char *tmpnam(s)
          char *s;

Return Codes -

     ptr     -  pointer to unique filename.
```

Usage Notes:

This function generates a unique filename each time it is called. *tmpnam(NULL)* creates a unique filename string and returns a pointer to an internal static array. *tmpnam(s)* stores the string in *s* as well as returning it as the value of the function. This function only generates a name, not a file.

Implementation Notes:

The parameter *s* must have room for at least *L_tmpnam* characters. The label *L_tmpnam* is defined in stdio.h.

At most *TMP_MAX* different names are guaranteed during the execution of the program.

ungetc

```
Declaration -

     #include "stdio.h"

     int ungetc(c, stream)
         int   c;
         FILE *stream;

Return Codes -

     EOF     -  error or end-of-file encountered.
     c       -  character pushed.
```

Usage Notes:

This function pushes one character back onto *stream*. The file associated with *stream* must be open for reading. The next character read from *stream* will be *c*.

Implementation Notes:

None.

Chapter 6. Low Level File I/O

close

```
Declaration -

     #include "fcntl.h"

     int close(fdesc)
         int fdesc;

Return Codes -

     0       -  stream successfully closed.
     EOF     -  error occurred. errno is set.
```

Usage Notes:

This functions closes the file associated with *fdesc*.

Implementation Notes:

None.

lseek

```
Declaration -

     #include "fcntl.h"

     long lseek(fdesc, offset, refpt)
          int   fdesc;
          long  offset;
          int   refpt;

Return Codes -

     nn      -  new offset from the beginning of the file, in bytes.
     EOF     -  error occurred. errno is set.
```

Usage Notes:

This function sets the file pointer to the requested position in the file associated with *fdesc*. The *offset* is relative to the given *refpt* value.

Table 6-1. Seek reference points.

Numerical value	Symbolic name	Description
0	SEEK_SET	Beginning of file
1	SEEK_CUR	Current position
2	SEEK_END	End of file

Implementation Notes:

No error is returned if you seek before the beginning of the file.

open

```
Declaration -

     #include "fcntl.h"

     int open(filename, mode, permission)
         char *filename;
         int mode;
         int permission;

Return Codes -

     fdesc   -  integer file descriptor.
     EOF     -  error occurred while opening the file. errno is set.
```

Usage Notes:

This function opens the file specified by filename for binary access.

The *mode* parameter specifies the type of access requested for the file.

Table 6-2. File open flags

Mode	Meaning
O_APPEND	Position file pointer at the end of the file before writing. This flag is only valid with either O_WRONLY or O_RDWR.
O_CREAT	If file doesn't exist, create and open a new file.
O_EXCL	If file exists, open returns an error. This flag is only valid with O_CREAT.
O_RDONLY	Open file for reading only. Don't specify O_RDWR or O_WRONLY if you are using this flag.
O_RDWR	Open file for reading and writing. Don't specify O_RDONLY or O_WRONLY if you are using this flag.
O_WRONLY	Open file for writing. Don't specify O_RDONLY or O_RDWR if you are using this flag.
O_TRUNC	If file exists truncate its length to zero. Use this flag with care since the file contents are lost.

You must at least specify one of the access mode flags, O_RDONLY, O_RDWR and O_WRONLY. The default action for an existing file is to open the file, the default action if the file doesn't exist is a failure.

Implementation Notes:

The *permission* parameter is currently ignored.

read

```
Declaration -

     #include "fcntl.h"

     int read(fdesc, buffer, buffersize)
         int          fdesc;
         char         *buffer;
         unsigned int buffersize;

Return Codes -

     nn      -  number of items read. 0 indicates end of file.
     EOF     -  error occurred. errno is set.
```

Usage Notes:

This function reads from the file associated with *fdesc buffersize* bytes into *buffer*. The actual number of bytes read is returned. A return value less than *buffersize* usually indicates an end of file condition.

Implementation Notes:

None.

tell

```
Declaration -

     #include "fcntl.h"

     int tell(fdesc)
         int fdesc;

Return Codes -

     nn      -  current offset from the beginning of the file, in bytes.
     EOF     -  error occurred. errno is set.
```

Usage Notes:

This function returns the current position of the file pointer relative to the beginning of the file associated with *fdesc*.

Implementation Notes:

None.

write

```
Declaration -

     #include "fcntl.h"

     int write(fdesc, buffer, length)
         int          fdesc;
         char        *buffer;
         unsigned int length;

Return Codes -

     nn      -  number of bytes written.
     EOF     -  error occurred. errno is set.
```

Usage Notes:

This function writes *length* bytes from the location given by *buffer* to the file associated with *fdesc*. The number of bytes written items is returned. A return value less than *nitems* usually indicates that no space is left on the device.

Implementation Notes:

If less bytes than requested are written *errno* is set and the number of bytes written is returned, not EOF. This case will occur when the volume being written to is full.

Chapter 7. Math Functions

abs

```
Declaration -

     #include "stdlib.h"

     int abs(val)
         int val;

Return Codes -

     nn      -  absolute value of val.
```

Usage Notes:

This function returns the absolute value of its argument.

div

```
Declaration -

     #include "stdlib.h"

     div_t div (num, denom)
         int num;
         int denom;

Return Codes -

     struct
     div_t   -  a structure containing the quotient and the remainder.
```

Usage Notes:

This function divides *num* by *denom*. The result is returned in a structure that contains two members, an integer quotient and an integer dividend. The structure is of type *div_t* and the quotient is named *quot* while the remainder is named *rem*.

labs

```
Declaration -

     #include "stdlib.h"

     long int labs(lval)
         long int lval;

Return Codes -

     nn      -  absolute value of lval.
```

Usage Notes:

This function returns the absolute value of its argument, which is type *long*.

ldiv

```
Declaration -

     #include "stdlib.h"

     ldiv_t ldiv (num, denom)
         long num;
         long denom;

Return Codes -

     struct
     ldiv_t   -  a structure containing the quotient and the remainder
```

Usage Notes:

This function divides *num* by *denom*. The result is returned in a structure that contains two members, a long integer quotient and an long integer dividend. The structure is of type *ldiv_t* and the quotient is named *quot* while the remainder is named *rem*.

Chapter 8. Memory Functions

memccpy

```
Declaration -

     #include "string.h"

     char * memccpy(buffer1, buffer2, c, cnt)
         void * buffer1;
         const void * buffer2;
         int c;
         size_t cnt;

Return Codes -

     nnnn    -  a pointer after value c in buffer1.
     NULL    -  cnt characters copied into buffer1.
```

Usage Notes:

This function copies *buffer2* to the location *buffer1* points to, but will stop copying after the value *c* has been copied into *buffer1* or if *cnt* characters are copied. If all *cnt* characters are copied then this functions returns NULL, otherwise, a pointer after the copied value *c* in *buffer1* is returned.

No care is taken of overlapping regions.

memchr

```
Declaration -

     #include "string.h"

     void * memchr(buffer, c, cnt)
         const void * buffer;
         int    c;
         size_t cnt;

Return Codes -

     ptr     -  a pointer to the first occurrence of c in string.
     NULL    -  if the character c is not found.
```

Usage Notes:

This function searches for the first occurrence of *c* in *string* and returns a pointer to it, or NULL if not found. At most *cnt* characters will be examined.

memcmp

```
Declaration -

     #include "string.h"

     int memcmp(buffer1, buffer2, cnt)
         const void * buffer1;
         const void * buffer2;
         size_t cnt;

Return Codes -

     0       -  buffer1 equals buffer2.
     >0      -  buffer1 is greater than buffer2.
     <0      -  buffer1 is less than buffer2.
```

Usage Notes:

This function compares *buffer1* and *buffer2* lexicographically. The comparison is ended if an inequality is found or *cnt* characters were compared.

memcpy

```
Declaration -

     #include "string.h"

     void * memcpy(buffer1, buffer2, cnt)
         void * buffer1;
         const void * buffer2;
         size_t cnt;

Return Codes -

     ptr     -  a pointer to buffer1.
```

Usage Notes:

This function copies *buffer2* to the location *buffer1* points to. Exactly *cnt* characters are copied.

No care is taken of overlapping regions.

memicmp

```
Declaration -

     #include "string.h"

     int memicmp(buffer1, buffer2, cnt)
         const void * buffer1;
         const void * buffer2;
         size_t cnt;

Return Codes -

     0       -  buffer1 equals buffer2.
     >0      -  buffer1 is greater than buffer2.
     <0      -  buffer1 is less than buffer2.
```

Usage Notes:

This is a case-insensitive version of memcmp. All upper case characters are translated to lower case before the comparison.

memmove

```
Declaration -

     #include "string.h"

     void * memmove(buffer1, buffer2, cnt)
         void * buffer1;
         const void * buffer2;
         size_t cnt;

Return Codes -

     ptr     -  a pointer to buffer1.
```

Usage Notes:

This function copies *buffer2* to the location *buffer1* points to. Exactly *cnt* characters are copied.

Overlapping regions are copied before being overwritten.

memset

```
Declaration -

     #include "string.h"

     void * memset(buffer, c, cnt)
         void * buffer;
         int    c;
         size_t cnt;

Return Codes -

     ptr     -  a pointer to buffer.
```

Usage Notes:

This function sets *cnt* characters of *buffer* to *c*.

swab

```
Declaration -

     void * swab(buffer1, buffer2, cnt)
         char * buffer1;
         char * buffer2;
         int cnt;

Return Codes -

     NONE
```

Usage Notes:

This function copies *cnt* bytes from the location pointed to by *buffer2* to the location *buffer1* points to, exchanging adjacent even and odd bytes. The *cnt* parameter should be even and non-negative. If *cnt* is odd and positive, *cnt*-1 bytes are copied. If *cnt* is a negative value, this routine does nothing.

Chapter 9. Print Functions

fprintf

```
Declaration -

     #include "stdio.h"

     int fprintf(stream, format, other_argument_1, ..., other_argument_n)
         FILE *stream;
         const char *format;
         some_type  other_argument_1;
         ...
         some_type  other_argument_n;

Return Codes -

     -nn     -  error during writing to stream.
     nn      -  number of characters written.
```

Usage Notes:

This function formats and prints a series of characters and values to the output *stream*.

A format specification has the following form:

%<*flags*><*width*><.*precision*> <h/l><*type*>

Table 9-1. Print flags

Flag	Meaning
+	Output value has a sign prefix.
-	Left justify the output.
blank	Prefix the output with a blank if the output is signed and positive.
#	Prefixes output with 0, 0x, 0X for the types o, x, X.

The *width* is a positive decimal integer and gives the minimal number of characters to be written for this field.

The *precision* gives the maximum number of characters printed for *type* s, the minimum number of digits to be printed for integer values and the number of digits after the decimal point for floating point values.

The prefix *h* is used with the types d, u, o, x and X to specify that the argument is a short int. The prefix *l* is used to specify that the argument is a long int with the types d, u, o, x and X, and that the argument is double instead of float with f, e, E, g and G types.

Table 9-2. Print type

Type Character	Output Format	Type of argument
c	Single character	char
s	Characters are printed until a null character is read or *precision* is reached.	pointer to char
d	Signed decimal integer	int
i	Signed decimal integer	int
u	Unsigned decimal integer	unsigned int
o	Unsigned octal integer	unsigned int
x	Unsigned hexadecimal integer, using 'abcdef'.	unsigned int
X	Unsigned hexadecimal integer, using 'ABCDEF'.	unsigned int
f	Signed floating point value.	double
e	Signed floating point value in exponential form.	double
E	Signed floating point value in exponential form. An 'E' introduces the exponent.	double
g	Signed floating point value in 'f' or 'e' form, whichever is more compact.	double
G	Signed floating point value in 'f' or 'E' form, whichever is more compact.	double
p	Pointer value	pointer to void
n	Number of characters written	pointer to int

Implementation Notes:

Printing out the value of "infinity" designated by the numeric coprocessor results in the string "INF" being printed to the output stream.

getsessid

```
Declaration -

     #include "stdio.h"

     unsigned long int getsessid()

Return Codes -

     nn      -  Session id of screen used by stdin, stdout and stderr.
```

Usage Notes:

This function returns the session id used by stdin, stdout and stderr. Use this session id for direct calls to the Session Manager. If the return value is zero, then no session has been created yet.

printf

```
Declaration -

     #include "stdio.h"

     int printf(format, other_argument_1, ..., other_argument_n)
         const char *format;
         some_type  other_argument_1;
         ...
         some_type  other_argument_n;

Return Codes -

     -nn     -  error during writing on stdout.
     nn      -  number of characters written.
```

Usage Notes:

This function formats and prints a series of characters and values on stdout. For more information refer to the description of *fprintf*.

setattr

```
Declaration -

     #include "stdio.h"

     void setattr(color)
         char color;

Return Codes -

     NONE
```

Usage Notes:

This function sets the color for the output to the screen.

setsessid

```
Declaration -

     #include "stdio.h"

     unsigned long int setsessid(newsessid)
         unsigned long int newsessid;

Return Codes -

     nn      -  Previous session id of screen used by stdin, stdout
                and stderr.
```

Usage Notes:

This function sets a new session id for stdin, stdout and stderr. It returns the previous session id. Use the returned id to restore I/O to the previous session. If the return value is zero, then no session has been created yet.

If *newsessid* is set to zero, then the session id used by the stdin, stdout, and stderr streams is closed.

Implementation notes

None

sprintf

```
Declaration -

     #include "stdio.h"

     int sprintf(string, format, other_argument_1, ..., other_argument_n)
         char *string;
         const char *format;
         some_type  other_argument_1;
         ...
         some_type  other_argument_n;

Return Codes -

     -nn     -  error during writing to the location given by string.
     nn      -  number of characters written.
```

Usage Notes:

This function formats and prints a series of characters and values to *string*. For more information refer to the description of *fprintf*.

vfprintf

```
Declaration -

     #include "stdio.h"
     #include "stdarg.h"

     int vfprintf(stream, format, arglist)
         FILE *stream;
         const char *format;
         va_list arglist;

Return Codes -

     -nn     -  error during writing to stream.
     nn      -  number of characters written.
```

Usage Notes:

This function formats and prints a series of characters and values to the output *stream*. It is identical to *fprintf*, except that the variable number of input arguments is specified in a va_list structure.

For more information refer to the description of *fprintf*.

vprintf

```
Declaration -

     #include "stdio.h"
     #include "stdarg.h"

     int vprintf(format, arglist)
         const char *format;
         va_list arglist;

Return Codes -

     -nn     -  error during writing to stream.
     nn      -  number of characters written.
```

Usage Notes:

This function formats and prints a series of characters and values on stdout. It is identical to *printf*, except that the variable number of input arguments is specified in a va_list structure.

For more information refer to the description of *fprintf*.

vsprintf

```
Declaration -

     #include "stdio.h"
     #include "stdarg.h"

     int vsprintf(string, format, arglist)
         char *string;
         const char *format;
         va_list arglist;

Return Codes -

     -nn     -  error during writing to stream.
     nn      -  number of characters written.
```

Usage Notes:

This function formats and prints a series of characters and values to *string*. It is identical to *sprintf*, except that the variable number of input arguments is specified in a va_list structure.

For more information refer to the description of *fprintf*.

Chapter 10. Program Control Functions

abort

```
Declaration -

     #include "stdlib.h"

     void abort(void)

Return Codes -

     NONE
```

Usage Notes:

This function terminates the program immediately without flushing buffers.

Implementation Notes:

The *abort* function uses a Task Halt SV Call to terminate. The return code value is reflected in the TCB.

atexit

```
Declaration -

     #include "stdlib.h"

     int atexit(function)
         void (* function)(void);

Return Codes -

     == 0    -  function successfully registered.
     != 0    -  function not registered.
```

Usage Notes:

This function registers the address of *function*. *function* will be called when *exit* is called. In case of multiple calls to *atexit* the function registered last will be executed first.

brkpt

```
Declaration -

     #include "stdlib.h"

     void brkpt(eax, ebx, ecx, edx, esi, edi)
         int eax;
         int ebx;
         int ecx;
         int edx;
         int esi;
         int edi;

Return Codes -

     NONE
```

Usage Notes:

This function transfers control to the debugger by executing an INT 2 instruction, thereby suspending execution of the program. The parameters are displayed in the EAX, EBX, ECX, EDX, ESI, and EDI registers respectively. These six registers are saved upon entry and restored upon exit.

Implementation Notes:

This is a non-ANSI C function.

exit

```
Declaration -

     #include "stdlib.h"

     void exit(rc)
         int rc;

Return Codes -

     NONE
```

Usage Notes:

This function calls all functions registered via an *atexit* call, closes every open file and calls *abort*.

longjmp

```
Declaration -

     #include "setjmp.h"

     void longjmp(env, retval)
         jmp_buf env;
         int retval;

Return Codes -

     NONE
```

Usage Notes:

The function *longjump* restores an environment previously set up by a call to *setjmp*. This includes returning control to a procedure at the point after which *setjmp* was invoked with a non-zero return code as specified by the second parameter, *retval*.

If *retval* is zero, then *longjmp* will return 1, as the returned value is the indicator that *setjmp's* return was in fact a result of *longjmp*.

paws

```
Declaration -

     #include "stdlib.h"

     void paws(status)
         int status;

Return Codes -

     NONE
```

Usage Notes:

If there is a session associated with your program., *paws()* writes a message to the screen and waits for a keystroke. Part of the message is the *status* parameter, displayed in decimal and hexadecimal.

setjmp

```
Declaration -

     #include "setjmp.h"

     int setjmp(env)
         jmp_buf env;

Return Codes -


      0         - setjmp function called directly.
     nn         - return after call from longjmp.
```

Usage Notes:

The function *setjmp* copies the current environment into the *jmp_buf* array. The current environment consists of register values that need to be saved for a correct return to the point after the *setjmp* call.

The function *setjmp* is used for non-local gotos. Typical usage is to return from perhaps several levels of procedure calls back to a top-level routine to handle errors.

The first return of *setjmp* returns zero. The second return to *setjmp* (ie: after a *longjmp* call) will return a non-zero value.

system

```
Declaration -

     #include "stdlib.h"

     int system(cmdstring)
         const char *cmdstring;

Return Codes -


     nn          - return code of executed command string.
     EOF         - error encountered in executing command string, errno set.
```

Usage Notes:

The *system()* call will suspend the current C program in order to run the program specified in *cmdstring*. When *cmdstring* terminates, the C program containing the *system()* call continues. *cmdstring* contains the filename of the program to be run as well as any command line arguments. Only ".XLD" executable files can be run by *system()*. There is no batch file support.

If EOF is returned by *system()*, the global variable *errno* can contain one of four values:

ENOENT program name not found.

ENOMEM could not allocate heap space.

EACCES Failure on process/task creation, for example, the program specified in the *system()* call required more process/task privilege than the caller had. Other possible failures include an error in either loading the program or parsing the associated .QID file.

EBADF could not create unique process/task name.

Implementation Notes:

The *system()* call performs the following actions:

- The current directory is searched for the filename specified as the first argument in *cmdstring*. If the file is not found, the PATH environment variable is used as the execution path. If the file is still not found, then an EOF is returned by *system()*.
- Create a unique process name. The algorithm to create a unique process name is same as that used by the Shell.
- Create a unique task name. The algorithm to create a unique task name is same as that used by the Shell.
- Search the .QIF file of the program (if present) for its program attributes.
- Create a process as a child of the caller's process using the process name determined above. System resource defaults are used for the process creation call, which can be overridden by entries in the .QIF file.

- Create a task in the created child process using the unique task name determined above and make the C program calling the *system()* function the faulter for the task created by *system* function.
- Load the program load module into the newly created task.
- Issue a GOTASK SVC to start to task.
- Wait for a fault message from the SVC Handler.
- Once the fault message arrives, save the *TCBerrcode* field of the faulted task.
- Delete the created child process from the system.
- Return the previously saved *TCBerrcode* field to the caller.

_exit

```
Declaration -

     #include "stdlib.h"

     void _exit(rc)
         int rc;

Return Codes -

     NONE
```

Usage Notes:

This function closes every open file and calls *abort*.

Chapter 11. Scan Functions

fscanf

```
Declaration -

     #include "stdio.h"

     int fscanf(stream, format, other_argument_1, ..., other_argument_n);
         FILE *stream;
         const char *format;
         pointer_to_some_type  other_argument_1;
         ...
         pointer_to_some_type  other_argument_n;

Return Codes -

     EOF     -  attempt to read past end of file.
     nn      -  number of fields assigned.
```

Usage Notes:

This function reads from *stream* into locations given by the pointers in the list of arguments. The string *format* controls the interpretation of the input fields.

A format specification has the following form:

%<*><*width*><h/l/L>*type*

An asterisk (*) after the percent sign (%) suppresses assignment of the next input field.

The *width* is a positive decimal integer and gives the maximum number of characters to be read from *stream*.

The prefix *h* indicates that the short version of the given type should be used. The prefix *l* stands for the long version to be used. The prefix *L* can precede the character conversions *e*,*f*, or *g* for a pointer to *long double*.

Table 11-1. Scan type

Type Character	Input accepted	Type of argument
c	character	pointer to char
s	Input is read until a whitespace is read or the specified *width* has been reached. A null character is appended to the input field.	pointer to char
d	decimal integer	pointer to int
i	integer. The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).	pointer to int
o	octal integer (with or without leading zero).	pointer to int
u	unsigned decimal integer	pointer to unsigned int
x	hexadecimal integer (with or without leading 0x or 0X).	pointer to int
e,f,g	floating point number. The input format for float is an optional sign, a string of numbers possibly containing a decimal point, and an optional exponent field containing an *e* or *E* followed by a possibly signed integer.	pointer to float
p	pointer value	pointer to int
n	Writes into the argument the number of characters output. No input is read. The item count is not incremented.	pointer to int
[...]	Characters are read up to the first character not in the set. The format string *[]...]* includes *]* in the set.	pointer to char
[^...]	Characters are read up to the first character listed in the set. The format string *[^]...]* includes *]* in the set.	pointer to char
%	literal %	No assignment is made.

Implementation Notes:

None.

scanf

```
Declaration -

     #include "stdio.h"

     int scanf(format, other_argument_1, ..., other_argument_n);
         const char *format;
         pointer_to_some_type  other_argument_1;
         ...
         pointer_to_some_type  other_argument_n;

Return Codes -

     EOF     -  attempt to read past end of file.
     nn      -  number of fields assigned.
```

Usage Notes:

This function reads from stdin into locations given by the pointers in the list of arguments. The string *format* controls the interpretation of the input fields. For more information refer to the description of *fscanf*.

sscanf

```
Declaration -

     #include "stdio.h"

     int sscanf(string, format, other_argument_1, ..., other_argument_n);
         const char *string;
         const char *format;
         pointer_to_some_type  other_argument_1;
         ...
         pointer_to_some_type  other_argument_n;

Return Codes -

     EOF     -  attempt to read past end of file.
     nn      -  number of fields assigned.
```

Usage Notes:

This function reads from *string* into locations given by the pointers in the list of arguments. The string *format* controls the interpretation of the input fields. For more information refer to the description of *fscanf*.

Chapter 12. Sort Functions

binsort

```
Declaration -

     #include "stdlib.h"

      int binsort(base, nel, elsize, nbin, convert_bin, iterations)
         void *  base;
         size_t  nel;
         size_t  elsize;
         size_t  nbin;
         int (*convert_bin)(const void *, int);
         int iterations;

Return Codes -

     TRUE    - Sort completed successfully
     FALSE   - Sort unsuccessful
```

Usage Notes:

This function uses the bin sort algorithm to sort an array of *nel* elements each of the size *elsize* starting at the location given by *base*. The parameter *nbin* specifies the number of different items that each position in the string could occupy.

The *convert_bin* parameter is the address of a function that returns a bin number given one of the items and an iteration count. The function format is a follows:

```
int convert_bin(element, iteration_count)
void       *element;
int        iteration_count;
```

The parameter *iterations* specifies the number of times that the bins need to be re-processed before the sort can complete. second element is less than the first.

The bin sort function can return FALSE under the following conditions: the parameter *nbin* is equal to zero or a call to *malloc()* from within the bin sort function returns NULL.

The bin sort is different from the other sort routines given in this chapter. Other sort routines in this chapter use comparisons of the whole object to determine what order they should be arranged in. The bin sort breaks objects down into smaller sections and compares the smaller sections. This is most easily illustrated using an example.

Example: Consider sorting the 4 numbers 99, 27, 37, 28 using a bin sort. A bin sort takes the numbers and compares them digit by digit. For these numbers,

number_of_iterations
2

convert_to_bin
returns the digit at the *index*th place in the number, i.e. 27 would return 2 if called on index 0, 7 if called on index 1 etc.

number_of_bins
10, as there are 10 digits 0-9.

Thus, the first pass would examine the **least** significant index, i.e. *number_of_iterations*-1. The numbers are arranged into 10 buckets depending on the values returned. The only buckets that would have anything in them are

7 27 37
8 28
9 99

The list is then collected again by concatenating bin 0 to bin 1 to bin 2 etc.. This gives a list of 27 37 28 99. The operation is then repeated, but for the next digit.

2 27 28
3 37
9 99

The list is now 27 28 37 99.

Example: This example shows how the bin sort can be used to sort an array of strings. The number of iterations is equivalent to the number of characters in the longest string, the number of buckets is equivalent to the maximum value character in the whole sequence. The conversion function returns the *index*th character from the string or 0 if the index is past the end.

```
unsigned convert_to_bin(a,index)
char **a;
unsigned index;
{
  if (strlen(*a)<index)
    return 0;
  return (*a)[index];
}

void binsort_string(base,number)
char *base[];
unsigned number;
{
  int maximum_length=0;
  char maxchar=0;
  char *p;
  int i;

  for (i=0;i<number;++i)
  {
    if (strlen(base[i])>maximum_length)
      maximum_length=strlen(base[i]);
    for (p=base[i];*p;++p)
    {
      if (*p>maxchar)
        maxchar=*p;
    }
  }
  sort_bin(base,number,sizeof(char *),maxchar+1,convert_to_bin,maximum_length);
}

int main(argc,argv)
int argc;
char *argv[];
{
  int i;

  binsort_string(argv+1,argc-1);
  for (i=1;i<argc;++i)
    printf("%d: %s\n",i,argv[i]);
  return 0;
}
```

Performance	n
Heap required	n+number_of_bins

Implementation Notes:

None

hsort

```
Declaration -

     #include "stdlib.h"

     void hsort(arrayptr, nel, elsize, comp)
         void *arrayptr;
         size_t  nel;
         size_t  elsize;
         int (*comp)(const void *, const void *);

Return Codes -

     NONE
```

Usage Notes:

This function uses the heap sort algorithm to sort an array of *nel* elements each of the size *elsize* starting at the location given by *arrayptr*.

The *comp* parameter is the address of a comparison function. This routine has two parameters, pointers to 2 elements of the array. It gives back a value reflecting the relation of the two elements. A return value less than zero means the first element is less than the second, zero means both elements are equal and a value greater than zero means the second element is less than the first. A sample comparision function and program follow:

Example

```
/*********************************************************************/
/* Example comparison function to be passed to above routine         */
/* It sorts on the basis of the first 4 characters ONLY              */
/*********************************************************************/
int comp_example(a,b)
char **a,**b;
{
  return strncmp(*a,*b,4);
}

int main(argc,argv)
int argc;
char *argv[];
{
  int i;
  if (argc==1)
  {
    printf("Usage: %s <string> <string> <string> <string> ...\n\n",argv[0]);
    printf("  prints out the strings on different lines in a sorted\n");
    printf("  order.  It is an unusual order as it based ONLY on the\n");
    printf("  first 4 characters.  If two strings have the same first\n");
    printf("  four characters, then they will appear in the order they\n");
    printf("  appeared on the command line\n");
  }
  else
  {
    sort_insertion(argv+1,argc-1,sizeof(argv[0]),comp_example);
    for (i=1;i<argc;++i)
      printf("%d: %s\n",i,argv[i]);
  }
  return 0;
}
```

The heap sort is not a stable sort.[1] However, it provides fair performance for large n and does not have any pathological bad cases.

Performance	$n \log_2(n)$
Heap required	n^0

Implementation Notes:

None

[1] For example, the strings "Theresa" "There" "Therapy" were sorted with a comparison function that compares the first 4 characters. The result will be "Theresa" "There" "Therapy" (i.e. the order in which they are in the array). Many other sort algoritms do not define the ordering under these circumstances.

inssort

```
Declaration -

     #include "stdlib.h"

     void inssort(arrayptr, nel, elsize, comp)
         void *arrayptr;
         size_t  nel;
         size_t  elsize;
         int (*comp)(const void *, const void *);

Return Codes -

     NONE
```

Usage Notes:

This function uses the insertion sort algorithm to sort an array of *nel* elements each of the size *elsize* starting at the location given by *arrayptr*. The elements are sorted in ascending order and the original elements are overwritten.

The *comp* parameter is the address of a comparison function. This routine has two parameters, pointers to 2 elements of the array. It gives back a value reflecting the relation of the two elements. A return value less than zero means the first element is less than the second, zero means both elements are equal and a value greater than zero means the second element is less than the first.

The insertion sort is a simple, reliable and fairly efficient sort for general usage. It is a common choice of sorting algorithm where sorting is not the limiting factor in execution time. One useful property of the insertion sort is that it is stable.[1]

Performance n^2

Heap required n^0

Implementation Notes:

None

msort

```
Declaration -

     #include "stdlib.h"

     void msort(arrayptr, nel, elsize, comp)
         void *arrayptr;
         size_t  nel;
         size_t  elsize;
         int (*comp)(const void *, const void *);

Return Codes -

     NONE
```

Usage Notes:

This function uses the merge sort algorithm to sort an array of *nel* elements each of the size *elsize* starting at the location given by *arrayptr*.

The *comp* parameter is the address of a comparison function. This routine has two parameters, pointers to 2 elements of the array. It gives back a value reflecting the relation of the two elements. A return value less than zero means the first element is less than the second, zero means both elements are equal and a value greater than zero means the second element is less than the first.

The merge sort is not a stable sort.[1] However, it does provide good performance for large n and does not have any pathological bad cases. It does require *elsize***nel* bytes of heap space.

Performance	$n \log_2(n)$
Heap required	n

Implementation Notes:

None

qsort

```
Declaration -

     #include "stdlib.h"

     void qsort(arrayptr, nel, elsize, comp)
         void *arrayptr;
         size_t  nel;
         size_t  elsize;
         int (*comp)(const void *, const void *);

Return Codes -

     NONE
```

Usage Notes:

This function sorts an array of *nel* elements each of the size *elsize* starting at the location given by *arrayptr*.

The *comp* parameter is the address of a comparison function. This routine has two parameters, pointers to 2 elements of the array. It gives back a value reflecting the relation of the two elements. A return value less than zero means the first element is less than the second, zero means both elements are equal and a value greater than zero means the second element is less than the first.

The quick sort is not a stable sort.[1] It does provide good performance for large n, as long as the array is not already sorted, and does not require any heap space.

Performance	$n \log_2(n)$ on average, n^2 if the list is already sorted
Heap required	n^0

Implementation Notes:

The sorting algorithm used is a modified version of the abstract Quicksort Algorithm found in Gehani, 'C: An Advanced Introduction' , page 129.

Chapter 13. String Functions

bcopy

```
Declaration -

     bcopy(string1, string2, len)
         char * string1;
         char * string2;
         int len;

Return Codes -

     NONE
```

Usage Notes:

This function copies *len* bytes from *string1* to *string2*. Note that the source and destination strings are reversed from *strcpy()*.

bcmp

```
Declaration -

     int bcmp(string1, string2, len)
         char * string1;
         char * string2;
         int len;

Return Codes -

     0       -  string1 and string2 are identical for length bytes.
     != 0    -  string1 and string2 are not identical.
```

Usage Notes:

This function compares byte string *string1* against byte string *string2*. The return value is zero if the strings are identical for *length* bytes. A non-zero return value indicates the strings are not identical.

bzero

```
Declaration -

     bzero(string, len)
         char * string;
         int len;

Return Codes -

     NONE
```

Usage Notes:

This function writes *len* bytes of zero (0) into the string pointed to by *string*.

ffs

```
Declaration -

     int ffs(i)
         int i;

Return Codes -

     nnnn - index of first bit in i starting from 1.
     0    - the argument has the value 0.
```

Usage Notes:

This function returns the index of the first bit set in *i* starting from 1. A zero is returned if *i* equals zero.

index

```
Declaration -

     #include "string.h"

     char * index(string1, c)
         const char * string1;
         int c;

Return Codes -

     ptr     -  a pointer to the first occurrence of character c in string1.
     0       -  character c not found in string1.
```

Usage Notes:

This function returns a pointer to the first occurrence of character *c* in *string1*. If *c* does not exist in *string1* then zero is returned.

rindex

```
Declaration -

     #include "string.h"

     char * rindex(string1, c)
         const char * string1;
         int c;

Return Codes -

     ptr     -  a pointer to the last occurrence of character c in string1.
     0       -  character c not found in string1.
```

Usage Notes:

This function returns a pointer to the last occurrence of character *c* in *string1*. If *c* does not exist in *string1* then zero is returned.

strcat

```
Declaration -

     #include "string.h"

     char * strcat(string1, string2)
         char * string1;
         const char * string1;

Return Codes -

     ptr     -  a pointer to the concatenated string (== string1).
```

Usage Notes:

This function appends *string2* to *string1*, overwriting the terminating null character of *string1*. The resulting string is null terminated.

strchr

```
Declaration -

     #include "string.h"

     char * strchr(string, c)
         const char * string;
         int    c;

Return Codes -

     ptr     -  a pointer to the first occurrence of c in string.
     NULL    -  if the character c is not found.
```

Usage Notes:

This function searches for the first occurrence of *c* in *string* and returns a pointer to it, or NULL if not found. The null ('\0') character is included in the search.

strcmp

```
Declaration -

     #include "string.h"

     int strcmp(string1, string2)
         const char * string1;
         const char * string2;

Return Codes -

     0       -  string1 equals string2.
     >0      -  string1 is greater than string2.
     <0      -  string1 is less than string2.
```

Usage Notes:

This function compares *string1* and *string2* lexicographically and case sensitive.

strcpy

```
Declaration -

     #include "string.h"

     char * strcpy(string1, string2)
         char * string1;
         const char * string2;

Return Codes -

     ptr     -  a pointer to the result string (== string1).
```

Usage Notes:

This function copies *string2* to the location *string1* points to. The terminating null ('\0') character is the last character copied.

strcspn

```
Declaration -

     #include "string.h"

     size_t strcspn(string1, string2)
         const char * string1;
         const char * string2;

Return Codes -

     nn      -  see below.
```

Usage Notes:

This function returns the length of the initial substring of *string1* that consists entirely of characters not in *string2*. The terminating null characters are not considered in the search. This function is the complement of strspn.

strdup

```
Declaration -

     #include "string.h"

     char * strdup(string1)
         char * string1;

Return Codes -

     ptr     -  pointer to duplicated string.
     NULL    -  failure in duplicating string.
```

Usage Notes:

This function duplicates the string pointed to by *string1* and returns a pointer to the duplciated string. Heap space is allocated to provide memory for the duplicated string. If the heap manager is unable to provide heap space for this function, NULL is returned.

strerror

```
Declaration -

     #include "string.h"

     char * strerror(errnum)
         int errnum;

Return Codes -

     str     -  pointer to error message string.
```

Usage Notes:

This function maps the error number specified by the *errnum* argument to an error message string appropriate for that error number. The return value of the function is a pointer to the error message string. Typically, *errnum* is an *errno* value.

If *strerror()* cannot map the error number to an error string, the string "(system defined error)" is returned.

stricmp

```
Declaration -

     #include "string.h"

     int stricmp(string1, string2)
         const char * string1;
         const char * string2;

Return Codes -

     0       -  string1 equals string2.
     >0      -  string1 is greater than string2.
     <0      -  string1 is less than string2.
```

Usage Notes:

This is a case-insensitive version of strcmp. All upper case characters are translated to lower case before the comparison.

strlen

```
Declaration -

     #include "string.h"

     size_t strlen(string)
         const char * string;

Return Codes -

     nn      -  the number of not '\0' characters.
```

Usage Notes:

The number of bytes in *string* not including the null character will be returned.

strlwr

```
Declaration -

     #include "string.h"

     char * strlwr(string)
         char * string;

Return Codes -

     ptr     - a pointer to string.
```

Usage Notes:

This function converts any upper case characters in *string* to lower case.

strncat

```
Declaration -

     #include "string.h"

     char * strncat(string1, string2, n)
         char * string1;
         const char * string2;
         size_t    n;

Return Codes -

     ptr     -  a pointer to the concatenated string (== string1).
```

Usage Notes:

Same as strcat, except that at most *n* characters will be appended. A null character terminates the resulting string.

strncmp

```
Declaration -

     #include "string.h"

     int strncmp(string1, string2, n)
         const char * string1;
         const char * string2;
         size_t n;

Return Codes -

     0       -  string1 equals string2.
     >0      -  string1 is greater than string2.
     <0      -  string1 is less than string2.
```

Usage Notes:

The comparison is ended if an inequality is found, a null character was encountered or *n* characters were compared.

strnicmp

```
Declaration -

     #include "string.h"

     int strnicmp(string1, string2, n)
         const char * string1;
         const char * string2;
         size_t  n;

Return Codes -

     0       -  string1 equals string2.
     >0      -  string1 is greater than string2.
     <0      -  string1 is less than string2.
```

Usage Notes:

This is the case-insensitive version of strncmp.

strncpy

```
Declaration -

     #include "string.h"

     char * strncpy(string1, string2, n)
         char * string1;
         const char * string2;
         size_t n;

Return Codes -

     ptr     -  a pointer to the result string (== string1).
```

Usage Notes:

This function copies exactly n characters, truncating or null-padding *string2*. The result is not null-terminated if the length of *string2* is greater or equal as *n*.

strpbrk

```
Declaration -

     #include "string.h"

     char * strpbrk(string1, string2)
         const char * string1;
         const char * string2;

Return Codes -

     ptr     -  a pointer to the first occurrence of any character from
                string2 in string1.
     NULL    -  if string1 and string2 have no characters in common.
```

Usage Notes:

This function looks for the first occurrence of any character from *string2* in *string1* and returns a pointer to it. The terminating null character is not considered in the search.

strrchr

```
Declaration -

     #include "string.h"

     char * strrchr(string1, c)
         const char * string;
         int    c;

Return Codes -

     ptr     -  a pointer to the last occurrence of c in string.
     NULL    -  if the character is not found.
```

Usage Notes:

This function searches for the last occurrence of *c* in *string* and returns a pointer to it, or NULL if not found. The null ('\0') character is included in the search.

strrev

```
Declaration -

     #include "string.h"

     char * strrev(string)
         char * string;

Return Codes -

     ptr     -  a pointer to the string.
```

Usage Notes:

This function reverses the character order of a string. The string pointer passed to the function is the same value that is returned by the function.

strspn

```
Declaration -

     #include "string.h"

     size_t strspn(string1, string2)
            const char * string1;
            const char * string2;

Return Codes -

     nn      -  see below.
```

Usage Notes:

This function returns the length of the initial substring of *string1* that consists entirely of characters in *string2*. The terminating null characters are not considered in the search. This function is the complement of strcspn.

strstr

```
Declaration -

     #include "string.h"

     char * strstr(string1, string2)
         const char * string1;
         const char * string2;

Return Codes -

     ptr     -  a pointer to the first occurrence of string2 in string1.
     NULL    -  if string2 is not found in string1.
```

Usage Notes:

This function searches *string1* for the first occurrence of *string2* and returns a pointer to it.

strtok

```
Declaration -

     #include "string.h"

     char * strtok(string1, string2)
         char * string1;
         const char * string2;

Return Codes -

     ptr     -  a pointer to the next token in string1.
     NULL    -  if no more tokens were found.
```

Usage Notes:

This function interprets *string1* as a series of tokens and *string2* as set of delimiters.

In the first call to strtok for a given *string*, strtok will return a pointer to the first token. To get the next token in *string1* call strtok with a NULL pointer as the first argument.

Implementation Notes:

All calls to strtok modify *string1*. A null character is inserted after each token found.

Example:

```
char *string = "Sein oder nicht sein, dass ist hier die Frage."

token = strtok(string, " ,.\n"); /* get first token: "Sein" */

while (token != NULL) {
  /* process the token */
  token = strtok(NULL, " ,.\n"); /* get next token */
  }
```

strtol

```
Declaration -

     #include "stdlib.h"

     long int strtol(string, endofstr, base)
         const char *  string;
         char ** endofstr;
         int     base;

Return Codes -

     nn      -  the value represented in string.
```

Usage Notes:

This function converts *string* into a long value. The format of the string expected is:

<whitespace> <sign> <0> <x> <digits> <rest>

If *endofstr* is not NULL, ** endofstr* points to rest. That means it points to the first character that stopped the scan.

If *base* is 0 the characters between sign and digits are used to determine the base. If a 0 is followed by an 'x' (or 'X') the base is 16. If a digit follows the 0, then the string is interpreted as an octal integer. If the first character is '1' - '9' the string is interpreted as a decimal integer.

If *base* is between 2 and 36, then it is used as the base of the number. *base* can also be specified as a character.

strtoul

```
Declaration -

     #include "stdlib.h"

     unsigned long int strtoul(string, endofstr, base)
         const char *  string;
         char ** endofstr;
         int     base;

Return Codes -

     nn      -  the value represented in string.
```

Usage Notes:

This function converts *string* into an unsigned long value. The format of the string expected is:

<whitespace> <sign> <0> <x> <digits> <rest>

If *endofstr* is not NULL, ** endofstr* points to rest. That means it points to the first character that stopped the scan.

If *base* is 0 the characters between sign and digits are used to determine the base. If a 0 is followed by an 'x' (or 'X') the base is 16. If a digit follows the 0, then the string is interpreted as an octal integer. If the first character is '1' - '9' the string is interpreted as a decimal integer.

If *base* is between 2 and 36, then it is used as the base of the number. *base* can also be specified as a character.

strupr

```
Declaration -

     #include "string.h"

     char * strupr(string)
         char * string;

Return Codes -

     ptr     - a pointer to string.
```

Usage Notes:

This function converts any lower case characters in *string* to upper case.

Chapter 14. Time Functions

asctime

```
Declaration -

     #include "time.h"

     char *asctime(time)
         const struct tm  *time;

Return Codes -

     ptr     -  a pointer to the character string result.
```

Usage Notes:

This function converts a *tm*-structure into a character string. The *time* value can be obtained from a call to *gmtime* or *localtime*.

The format of the string returned is the following:

Sun Dec 31 11:14:54 1963\n\0

The string is exactly 26 characters long.

Implementation Notes:

The *asctime* and *ctime* functions are using one statically allocated buffer for the return string. Following calls destroy the result of the previous call.

clock

```
Declaration -

     #include "time.h"

     clock_t clock(void)

Return Codes -

     nn      -  returns the amount of CPU time used by a task.
```

Usage Notes:

The *clock* function returns the amount of CPU time used by the calling task. To measure the amount of processor time used by a program, the *clock()* function should be called at the beginning of the program, and that return value should be subtracted from the return value of future calls to the *clock* function.

To find the number of seconds used by a program, divide the return value of the *clock()* function by the value of the constant macro **CLOCKS_PER_SEC** (defined in the header file **time.h**).

Implementation Notes:

The *clock()* function requires a CMOS real time clock and the CPU time support within the SVC Handler in order to work. If a CMOS real time clock is not present or CPU time functionality has been omitted from the SVC Handler (by specifing the SLEEP "SYSPARM NOCMOS" or "SYSPARM NOCPUTIME" commands within a CP/Q build file), *clock()* returns zero.

ctime

```
Declaration -

     #include "time.h"

     char *ctime(tmval)
         const time_t *tmval;

Return Codes -

     ptr     -  a pointer to the character string result.
```

Usage Notes:

This function converts a time value, usually obtained from a call to the *time* function, to a character string.

The format of the string returned is the following:

Sun Dec 31 11:14:54 1963\n\0

The string is exactly 26 characters long.

Implementation Notes:

The *asctime* and *ctime* functions are using one statically allocated buffer for the return string. Following calls destroy the result of the previous call.

difftime

```
Declaration -

     #include "time.h"

     time_t difftime(time2, time1)
        time_t time2;
        time_t time1;

Return Codes -

     nn      -  difference in seconds from time1 to time2
```

This functions returns the difference between two times in seconds. The values for *time1* and *time2* are usually obtained using the *time* function.

gmtime

```
Declaration -

     #include "time.h"

     struct tm  *gmtime(tmval)
         const time_t *tmval;

Return Codes -

     ptr     -  a pointer to a structure.
```

Usage Notes:

This function converts a time value, usually obtained from a call to the *time* function, to a structure representing Greenwich Mean Time.

The value of the *tmval* parameter is the seconds elapsed since 00:00:00, January 1, 1970, Greenwich mean time.

Implementation Notes:

The *gmtime* and *localtime* functions are using one statically allocated buffer for the return structure. Following calls destroy the result of the previous call.

localtime

```
Declaration -

     #include "time.h"

     struct tm  *localtime(tmval)
         const time_t *tmval;

Return Codes -

     ptr     -  a pointer to a structure.
```

Usage Notes:

This function converts a time value, usually obtained from a call to the *time* function, to a structure, representing the local time.

The value of the *tmval* parameter is the seconds elapsed since 00:00:00, January 1, 1970, Greenwich mean time.

Implementation Notes:

The *gmtime* and *localtime* functions are using one statically allocated buffer for the return structure. Following calls destroy the result of the previous call.

Static information is used for the information about the time zone.

mktime

```
Declaration -

     #include "time.h"

     time_t mktime(time)
        struct tm *time;

Return Codes -

     nn      - the number of seconds elapsed since 00:00:00, January 1,
               1970, Greenwich mean time.
     -1      - if time cannot be represented.
```

Usage notes:

This function converts the local time contents of structure *time* into the time in seconds since January 1, 1970, Greenwich mean time - the same representation used by the function *time*. This function increases *tm_min* for every 60 seconds subtracted from *tm_sec* until *tm_sec* is in the interval [0,59]. This function also increases *tm_hour* for every 60 minutes subtracted from *tm_min* until *tm_min* is in the range [0,23]. This correction is performed in a similar way through to *tm_year*.

If structure member *is_dst* equals zero, then Daylight Savings Time is assumed not to be in effect for the particular time passed to *mktime()*. If structure member *is_dst* is greater than zero, then Daylight Savings Time is assumed to be in effect for the particular time passed to *mktime()*. If structure member *is_dst* is less than zero, then *mktime()* attempts to determine whether Daylight Savings Time was in effect for that particular time.

Upon successful completion of this call, the structure values *tm_wday* and *tm_yday* are computed and returned to the caller in the input structure.

strftime

```
Declaration -

     #include "time.h"

     size_t strftime(buffer, bufmax, format, time)
        char *buffer;
        size_t bufmax;
        const char *format;
        const struct tm *time;

Return Codes -

     0       - if more than bufmax characters were produced.
     nn      - number of characters written to buffer, excluding the NULL
               character.
```

Usage Notes:

This function formats date and time information from *time* into *buffer* according to the *format* string. The *format* string is analogous to a *printf* format. No more than *bufmax* characters are placed into *buffer*. Options allowed in the *format* string are described on the next page.

Please note that the NULL character is appended onto the string pointed to by *buffer*. The NULL character should be included in the *bufmax* count.

Table 14-1. Strftime format tokens	
Token	**Output Format**
%a	Abbreviated weekday name.
%A	Full weekday name.
%b	Abbreviated month name.
%B	Full month name.
%c	Local date and time representation.
%d	Day of the month (01-31).
%H	Hour (24-hour clock) (00-23).
%I	Hour (12-hour clock) (01-12).
%j	Day of the year (001-366).
%m	Month (01-12).
%M	Minute (00-59).
%p	AM or PM
%S	Second (00-61).
%U	Week number of the year using Sunday as first day of the week (00-53).
%w	Weekday starting with Sunday (0-6).
%W	Week number of the year using Monday of first day of the week (00-53).
%x	Local date representation.
%X	Local time representation.
%y	Year without century (00-99).
%Y	Year with century.
%Z	Time zone name.
%%	%.

Spaces, special character constants (with the preceeding backslash), as well as alphanumeric characters are allowed in the *format* string and are passed directly into *buffer*.

time

```
Declaration -

     #include "time.h"

     time_t time(tmval)
        time_t *tmval;

Return Codes -

     nn      -  the number of seconds elapsed since 00:00:00, January 1,
                1970, Greenwich mean time.
```

Usage Notes:

This function returns the time in seconds since January 1, 1970, Greenwich mean time. The same value is stored in the location given by *tmval* unless *tmval* equals NULL.

Chapter 15. Translate Functions

atof

```
Declaration -

     #include "stdlib.h"

     double atof(string)
         const char *string;

Return Codes -

     nn      -  the floating point value that corresponds to string.
```

Usage Notes:

The format of *string* is the following:

<whitespace> <sign> digits <decimal point> <digits> ('e'|'E' sign digits)

The first character that does not conform to the format above stops the scan.

atoi

```
Declaration -

     #include "stdlib.h"

     int atoi(string)
         const char *string;

Return Codes -

     nn      -  the integer value that corresponds to string.
```

Usage Notes:

The format of *string* is the following:

<whitespace> <sign> <whitespace> digits

The first character that cannot be interpreted as an integer value stops the scan.

atol

```
Declaration -

     #include "stdlib.h"

     long int atol(string)
         const char *string;

Return Codes -

     nn      -  the long integer value that corresponds to string.
```

Usage Notes:

The format of *string* is the following:

<whitespace> <sign> <whitespace> digits

The first character that cannot be interpreted as a long value stops the scan.

itoa

```
Declaration -

     #include "stdlib.h"

     char *itoa(num, string, radix)
         int num;
         char *string;
         int radix;

Return Codes -

     ptr     -  a pointer to the converted string.
     NULL    -  conversion could not be performed.
```

Usage Notes:

This function converts an integer value to a character string using the number base defined by *radix*. The range of *radix* is 2 to 36. A *radix* of 0 is not allowed. The function returns a pointer to the converted string as a function value as well as in the variable *string*

tolower,_tolower

```
Declaration -

     #include "ctype.h"

     int tolower(c)
         int c;

     int _tolower(c)
         int c;

Return Codes -

     nn      -  the lowercase equivalent of c.
```

Usage Notes:

The function *tolower* returns the corresponding lowercase letter if *c* is an uppercase letter, otherwise *c* is returned.

The macro *_tolower* returns the corresponding lowercase letter by requiring that the parameter *c* is an uppercase letter.

Implementation Notes:

The difference between *tolower* and *_tolower* is that *tolower* is a function and *_tolower* is a macro. The macro takes less time to execute because it omits the call and return of a function definition. However, the macro requires that the character passed to it is an uppercase character because the macro performs a bitwise OR to the character. Only use *_tolower* when you are sure that the character being passed to it is an uppercase character.

toupper,_toupper

```
Declaration -

     #include "ctype.h"

     int toupper(c)
         int c;

     int _toupper(c)
         int c;

Return Codes -

     nn      -  the uppercase equivalent of c.
```

Usage Notes:

The function *toupper* returns the corresponding uppercase letter if *c* is an lowercase letter, otherwise *c* is returned.

The macro *_toupper* returns the corresponding uppercase letter by requiring that the parameter *c* is an lowercase letter.

Implementation Notes:

The difference between *toupper* and *_toupper* is that *toupper* is a function and *_toupper* is a macro. The macro takes less time to execute because it omits the call and return of a function definition. However, the macro requires that the character passed to it is an uppercase character because the macro performs a bitwise AND to the character. Only use *_toupper* when you are sure that the character being passed to it is a lowercase character.

Chapter 16. Type Functions

isalnum,_isalnum

```
Declaration -

     #include "ctype.h"

     int isalnum(c)
         int c;

     int _isalnum(c)
         int c;

Return Codes -

     != 0    -  c is alphanumeric ('A'-'Z', 'a'-'z' or '0'-'9').
     == 0    -  c is not alphanumeric.
```

Usage Notes:

The difference between *isalnum* and *_isalnum* is that *isalnum* is a macro and *_isalnum* is a function.

Implementation Notes:

The *isalnum* macro only evaluates its parameter once to avoid possible side-effects.

isalpha,_isalpha

```
Declaration -

     #include "ctype.h"

     int isalpha(c)
         int c;
     int _isalpha(c)
         int c;

Return Codes -

     != 0    -  c is a letter ('A'-'Z' or 'a'-'z').
     == 0    -  c is not a letter.
```

Usage Notes:

The difference between *isalpha* and *_isalpha* is that *isalpha* is a macro and *_isalpha* is a function.

Implementation Notes:

The *isalpha* macro only evaluates its parameter once to avoid possible side-effects.

isascii,_isascii

```
Declaration -

     #include "ctype.h"

     int isascii(c)
         int c;
     int _isascii(c)
         int c;

Return Codes -

     != 0    -  c is a valid ASCII character.
     == 0    -  c is not a valid ASCII character.
```

Usage Notes:

The difference between *isascii* and *_isascii* is that *isascii* is a macro and *_isascii* is a function. A valid ASCII character has a character code less than 128(dec).

Implementation Notes:

The *isascii* macro only evaluates its parameter once to avoid possible side-effects.

iscntrl,_iscntrl

```
Declaration -

     #include "ctype.h"

     int iscntrl(c)
         int c;

     int _iscntrl(c)
         int c;

Return Codes -

     != 0    -  c is a control character (hex '00'-'1F' or '7F').
     == 0    -  c is not a control character.
```

Usage Notes:

The difference between *iscntrl* and *_iscntrl* is that *iscntrl* is a macro and *_iscntrl* is a function.

Implementation Notes:

The *iscntrl* macro only evaluates its parameter once to avoid possible side-effects.

isdigit,_isdigit

```
Declaration -

     #include "ctype.h"

     int isdigit(c)
         int c;

     int _isdigit(c)
         int c;

Return Codes -

     != 0    -  c is a digit ('0'-'9').
     == 0    -  c is not a digit.
```

Usage Notes:

The difference between *isdigit* and *_isdigit* is that *isdigit* is a macro and *_isdigit* is a function.

Implementation Notes:

The *isdigit* macro only evaluates its parameter once to avoid possible side-effects.

isgraph,_isgraph

```
Declaration -

     #include "ctype.h"

     int isgraph(c)
         int c;

     int _isgraph(c)
         int c;

Return Codes -

     != 0    -  c is a printable character (hex '21'-'7E').
     == 0    -  c is not a printable character.
```

Usage Notes:

The difference between *isgraph* and *_isgraph* is that *isgraph* is a macro and *_isgraph* is a function.

The space character is not included in the valid character set.

Implementation Notes:

The *isgraph* macro only evaluates its parameter once to avoid possible side-effects.

islower,_islower

```
Declaration -

     #include "ctype.h"

     int islower(c)
         int c;

     int _islower(c)
         int c;

Return Codes -

     != 0    -  c is a lowercase character ('a'-'z').
     == 0    -  c is not a lowercase character.
```

Usage Notes:

The difference between *islower* and *_islower* is that *islower* is a macro and *_islower* is a function.

Implementation Notes:

The *islower* macro only evaluates its parameter once to avoid possible side-effects.

isprint,_isprint

```
Declaration -

     #include "ctype.h"

     int isprint(c)
         int c;

     int _isprint(c)
         int c;

Return Codes -

     != 0    -  c is a printable character (hex '20'-'7E').
     == 0    -  c is not a printable character.
```

Usage Notes:

The difference between *isprint* and *_isprint* is that *isprint* is a macro and *_isprint* is a function.

The space character is included in the valid character set.

Implementation Notes:

The *isprint* macro only evaluates its parameter once to avoid possible side-effects.

ispunct,_ispunct

```
Declaration -

     #include "ctype.h"

     int ispunct(c)
         int c;

     int _ispunct(c)
         int c;

Return Codes -

     != 0    -  c is a punctuation character.
     == 0    -  c is not a punctuation character.
```

Usage Notes:

The difference between *ispunct* and *_ispunct* is that *ispunct* is a macro and *_ispunct* is a function.

Punctuation characters are all printable characters but space, lowercase letters, uppercase letters and digits (hex '21'-'2F', '3A'-'40', '5B'-'60' or '7B'-'7E').

Implementation Notes:

The *ispunct* macro only evaluates its parameter once to avoid possible side-effects.

isspace,_isspace

```
Declaration -

     #include "ctype.h"

     int isspace(c)
         int c;

     int _isspace(c)
         int c;

Return Codes -

     != 0    -  c is a white-space character (hex '09'-'0D' or '20').
     == 0    -  c is not a white-space character.
```

Usage Notes:

The difference between *isspace* and *_isspace* is that *isspace* is a macro and *_isspace* is a function.

Implementation Notes:

The *isspace* macro only evaluates its parameter once to avoid possible side-effects.

issupv

```
Declaration -

     #include "ctype.h"

     int issupv()

Return Codes -

     TRUE (1)-  code currently running at supervisor mode.
     FALSE(0)-  code currently running at user mode.
```

Usage Notes:

The function returns information concerning the privilege level of the currently executing code.

Implementation Notes:

issuov() is a function call, not a macro.

isupper,_isupper

```
Declaration -

     #include "ctype.h"

     int isupper(c)
         int c;

     int _isupper(c)
         int c;

Return Codes -

     != 0    -  c is a uppercase character ('A'-'Z').
     == 0    -  c is not a uppercase character.
```

Usage Notes:

The difference between *isupper* and *_isupper* is that *isupper* is a macro and *_isupper* is a function.

Implementation Notes:

The *isupper* macro only evaluates its parameter once to avoid possible side-effects.

isxdigit,_isxdigit

```
Declaration -

     #include "ctype.h"

     int isxdigit(c)
         int c;

     int _isxdigit(c)
         int c;

Return Codes -

     != 0    -  c is a hexadecimal digit ('A'-'F', 'a'-'f' or '0'-'9').
     == 0    -  c is not a hexadecimal digit.
```

Usage Notes:

The difference between *isxdigit* and *_isxdigit* is that *isxdigit* is a macro and *_isxdigit* is a function.

Implementation Notes:

The *isxdigit* macro only evaluates its parameter once to avoid possible side-effects.

Chapter 17. Variable Argument Functions

va_arg

```
Declaration -

     #include "stdarg.h"

     type va_arg(ap, type)
         va_list ap;

Return Codes -

     nn      -  the next argument on the variable length argument list.
```

Usage Notes:

The *va_arg* macro expands to an expression that has the type and value of the next argument in the variable argument list. The *ap* parameter must have been initialized by *va_start*.

type specifies the type of the argument to be fetched from the variable argument list.

va_end

```
Declaration -

     #include "stdarg.h"

     void va_end(ap)
         va_list ap;

Return Codes -

     NONE
```

Usage Notes:

The *va_end* macro allows a normal return from the function that initialized the variable argument list that invoked *va_start*. After invoking *va_end*, the variable argument list should not be accessed by the program with subsequent calls to *va_arg*.

va_start

```
Declaration -

     #include "stdarg.h"

     void va_start(ap, parmN)
         va_list ap;

Return Codes -

     NONE
```

Usage Notes:

The *va_start* macro initializes *ap* for subsequent use by the variable argument macros *va_arg* and *va_end*.

The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the , ...).

Chapter 18. Miscellaneous Functions

assert

```
Declaration -

     #include "assert.h"

     void assert(val)
         int val;

Return Codes -

     None
```

Usage Notes:

This function checks the expression *val*. If *val* is zero, a message is printed to stdout that includes the current line number of the source line, and the source file name, then *assert* calls *abort*. This function is useful for debugging purposes.

By defining the macro *NDEBUG* before including the header assert.h, the *assert* function will be disabled. This is useful in turning off debugging output for ship-level code.

bsearch

```
Declaration -

     #include "stdlib.h"

     void *bsearch(key, array, nel, elsize, comp)
         const void *key;
         const void *array;
         size_t nel;
         size_t elsize;
         int (*comp)(const void *,const char *);

Return Codes -

     NULL    -  key not found.
     ptr     -  address of key.
```

Usage Notes:

This function searches for *key* in a previously sorted array, beginning at the location given by *array*. The array consists of *nel* elements each of the size *elsize*. The *bsearch* function returns the address of the first element in the array where the comparison function *comp* gives back zero, when comparing the array element and *key*.

The *comp* parameter is the address of a comparison function. This routine has two parameters, pointers to 2 elements of the array. It gives back a value reflecting the relation of the two elements. A return value less than zero means the first element is less than the second, zero means both elements are equal and a value greater than zero means the second element is less than the first.

Implementation Notes:

The algorithm used is the binary search algorithm found in Kernighan & Ritchie book.

getopt

```
Declaration -

     int getopt(argc, argv, optstring)
         int argc;
         char **argv;
         char *optstring;

     extern char *optarg;
     extern int optind;

Return Codes -

     c       -  next option letter that matches a letter in optstring
     ?       -  option letter found that is not inlcluded in optstring
     EOF     -  all options processed, or "--" found in optstring
```

Usage Notes:

The purpose of the *getopt()* function is to aid programs in parsing command line arguments. The *argc* and *argv* function parameters are the argument count and argument array as passed to the *main()* function of a C program. The *optstring* parameter is a string of flag letters that are recognized by the program. If a flag letter is followed by a colon (':'), then that flag has an argument. Each command line argument to be recognized by *getopt()* as a command line flag must be preceded by a dash (ie: "-a").

getopt() returns the next option letter in *argv* that matches a flag letter in "hp1.optstring. A message is printed to stderr and a question mark('?') is returned by *getopt()* when an unrecognized flag letter is found in *argv*. An unrecognized flag is one that does not appear in *optstring*.

The external global variable *optarg* is set after a call to *getopt()* to point to the start of the argument string if the flag letter found was followed by a colon. If the flag letter found does not take an argument, then *optarg* is NULL.

The external global variable *optind* is set to the next index of the *argv* array to be processed. *optind* should be initialized to zero before the first call to *getopt()*.

When all options are processed, *getopt()* returns EOF. *getopt()* also returns EOF for the special option "--".

Example:

The sample code fragment in Figure 18-1 on page 18-4 demonstrates the use of the *getopt()* function. The code fragment is that of a command which accepts the command line flags **c** and **r**, and the flag **f**, which requires an argument.

```
#include <stdio.h>

extern int optind;
extern char *optarg;

int main(int argc, char **argv)
{
int c;

while ((c = getopt(argc, argv, "crf:")) != EOF)
    {
    switch(c)
        {
        case 'c':
            /* do processing for the "-c" parameter */
            do_cflag();
            break;
        case 'r':
            /* do processing for the "-r" parameter */
            do_rflag();
            break;
        case 'f':
            /* do processing for the "-f" parameter */
            f_arg = optarg;
            do_fflag();
            break;
        case '?':
            /* unrecognized option, print error message */
            fprintf(stderr,"Error: incorrect argument\n");
        }
    }
.
.
.

}
```

Figure 18-1. Example code fragment for getopt().

perror

```
Declaration -

     #include "stdio.h"

     void perror(string)
         const char *string;

Return Codes -

     NONE
```

Usage Notes:

This function prints an error message on *stderr*. The message consists of two parts. First *string* is printed, then a message describing the current value of the global variable *errno* is written on *stderr*.

raise

```
Declaration -

     #include "signal.h"

     int raise(sig)
         int *sig;

Return Codes -

     != 0    -  signal out of range.
     == 0    -  signal handler invoked.
```

Usage Notes:

This function invokes a function previously registered as the signal handler for *sig*. To register a function use *signal*.

Signals are initialized according to Table 18-1 on page 18-8.

Signals SIGKILL, SIGSTOP and SIGCONT cannot be ignored.

Implementation Notes:

The *raise* function makes use of the signal facilities provided by the SVC Handler. See *Systems Programming Library Volume 3: SVC Handler* for more information concerning the implementation of signals.

Please note that nested signals are not handled by this implementation.

rand

```
Declaration -

     #include "stdlib.h"

     int rand()

Return Codes -

     nn      -  a pseudo-random number.
```

Usage Notes:

This function returns a pseudo-random integer in the range 0 to 32767.

signal

```
Declaration -

     #include "signal.h"

     void (* signal(sig, function))
         int  sig;
         void (* function)(int);

Return Codes -

     SIG_ERR -  signal out of range.
     fptr    -  address of previous signal handler for this signal.
```

Usage Notes:

This function registers a signal handler, *function*, for a specific signal, *sig*. Use the function *raise* to signal the signal handler function.

Signals are initialized according to Table 18-1.

If a specific signal (other than SIGKILL, SIGSTOP and SIGCONT) should be ignored you have to use *SIG_IGN* as *function*.

Signals SIGKILL, SIGSTOP and SIGCONT cannot be ignored. If this function is called in an attempt to ignore these signals, then the function call will return failure and set *errno* to EINVAL.

Implementation Notes:

The *signal* function makes use of the signal facilities provided by the SVC Handler. See *Systems Programming Library Volume 3: SVC Handler* for more information concerning the implementation of signals.

Please note that nested signals are not handled by this implementation.

Table 18-1 (Page 1 of 2). Signal Names

Signal	Number	Default	Meaning or Use
SIGHUP	1	terminate	the control terminal has “hung up” or been closed.
SIGINT	2	terminate	interrupt key (i.e. Ctrl_Break) has been pressed on the keyboard.
SIGQUIT	3	terminate	“quit” key has been pressed on the keyboard.
SIGILL	4	terminate	illegal instruction has been attempted.
SIGTRAP	5	terminate	trace trap (not used in CP/Q).
SIGABRT	6	terminate	abort program.
SIGEMT	7	terminate	EMT instruction (not used in CP/Q).
SIGFPE	8	terminate	floating point error, also integer divide by 0.
SIGKILL	9	terminate	“kill” the task, cannot be caught or masked.
SIGBUS	10	terminate	bus error (not used in CP/Q).
SIGSEGV	11	terminate	segmentation violation - in CP/Q this can be used to trap page faults.
SIGSYS	12	terminate	bad argument to a system call.
SIGPIPE	13	terminate	write to a pipe that has no reader
SIGALRM	14	terminate	alarm timer signal

Table 18-1 (Page 2 of 2). Signal Names

Signal	Number	Default	Meaning or Use
SIGTERM	15	terminate	software termination
SIGURG	16	ignore	an urgent condition is present on a socket.
SIGSTOP	17	stop task	stop the task, cannot be caught or ignored. On CP/Q, this is the same as a STOP_TASK SVC.
SIGTSTP	18	stop task	"stop" has been pressed on the keyboard (i.e. "Pause" in IBM PC terminology) - the task is stopped. On CP/Q, this is the same as a STOP_TASK SVC, except that it may alternatively be caught by the task.
SIGCONT	19	restart task	continue the task after SIGSTP. On CP/Q, this is the same as a GO_TASK SVC, except that it may alternatively be caught by the task.
SIGCHLD	20	ignore	child task has stopped. On CP/Q, if no action is defined for this signal, the parent will receive a message generated by the SVC Handler.
SIGTTIN	21	stop task	read from the control terminal from a background task.
SIGTTOU	22	stop task	write to the control terminal from a background task.
SIGIO	23	ignore	I/O may be performed on a file descriptor (some versions of UNIX provide this to make possible a form of asynchronous I/O) (not used in CP/Q).
SIGXCPU	24	terminate	the task has exceeded its CPU time limit.
SIGXFSZ	25	terminate	the task has exceeded its file size limit (not used in CP/Q).
SIGVTALRM	26	terminate	virtual time alarm (not used in CP/Q).
SIGPROF	27	terminate	profiling time alarm (not used in CP/Q).
SIGWINCH	28	ignore	screen or screen window size has changed.
SIGINFO	29	terminate	user information.
SIGUSR1	30	terminate	user defined signal 1.
SIGUSR2	31	terminate	user defined signal 2.

srand

```
Declaration -

     #include "stdlib.h"

     void srand(seed)
         unsigned int seed;

Return Codes -

     NONE
```

Usage Notes:

This function sets the starting point for generating pseudo-random numbers.

Chapter 19. Technical Reference

General Organization of a C Program

A CP/Q C program is assumed to have four major parts: code, data, stack, and a heap. The organization of which is shown in Figure 19-1:

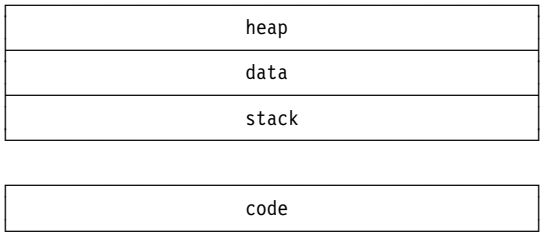


Figure 19-1. General organization of a C program

The reason that the code area is separate from the data, heap and stack is because of CP/Q memory management: the code area will be defined as a code object, and the data, heap and stack areas will be defined as one or more data objects.

The stack is an area of memory that grows down, that is, from higher memory address to lower memory addresses. The heap is an area of memory that grows upwards, from lower memory addresses to higher memory addresses. The memory addresses reserved for code and data are static - they do not increase or decrease in size. There is a concern that the stack could expand down far enough so that the stack exceeds the allocated stack area. The way to solve this problem is to use a CP/Q memory feature - force a memory page not present before the stack area in the address space. See Figure 19-2 for an illustration of the stack area.

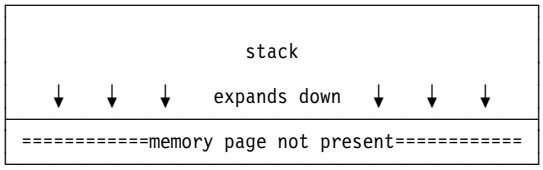


Figure 19-2. Stack and heap organization

If the stack approaches the not present memory page, the stack will try to write on the not present page and cause a page fault. The page fault handler will fault the task. The C program must be run again with a larger stack space. A larger stack space can be specified in a .QIF file for the program. See *Systems Programming Library Volume 12: Loader* for more information on .QIF files.

Up until now, we have been discussing a very simple model of a C program. We will now move from this simple model to a more flexible model that utilizes the memory management features of CP/Q. First we must back up and discuss the linear memory layout provided by CP/Q memory management. There are three memory classes provided by the CP/Q Memory Manager:

Global objects These objects are located in the global class and appear uniformly in all address spaces. Such objects are the CP/Q system kernel.

Shared objects These objects are located in the common class. When created, their range of linear offsets are reserved in all address spaces. Such objects include the code and data portion of a shared library routine.

Private objects These objects appear exclusively in a single address space. Typical objects in this class include the stack, heap, and private copies of code and data.

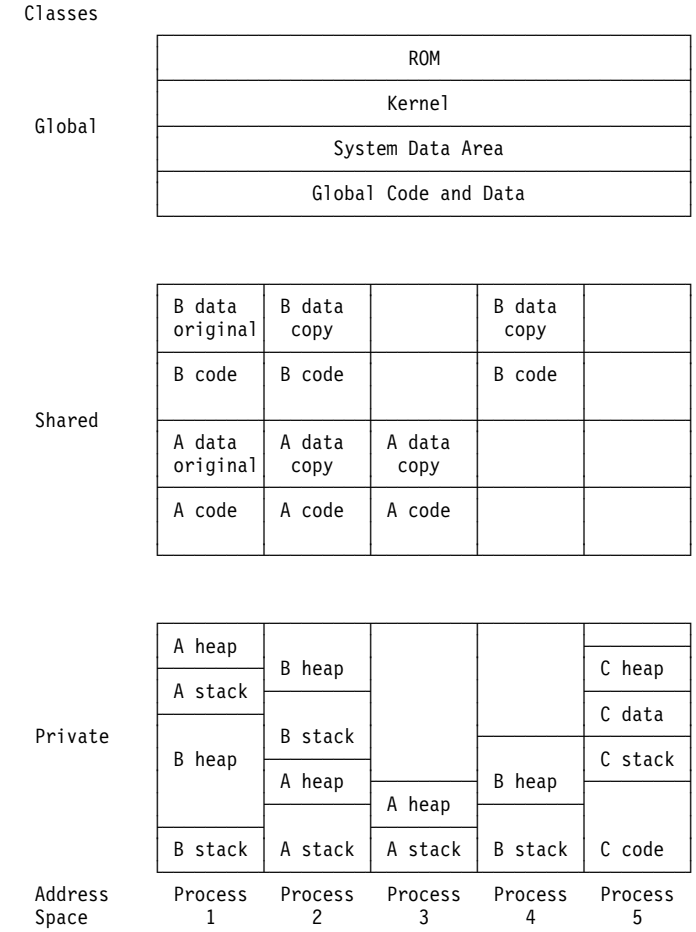


Figure 19-3. Example address spaces in CP/Q

The illustration in Figure 19-3 represents what each address space "sees", and may not necessarily be how the physical memory is laid out underneath. For more information on CP/Q memory objects and memory mangement, please see *Systems Programming Library Volume 4: Memory Manager*. The reason for the above discussion on CP/Q memory management is to bring out the following two points with respect to C programs under CP/Q:

- There can be multiple instances of heap and stack structures within an address space.
- Heap and stack structures will reside in the private class.

We should note here that the memory management of the initial heap and stack is taken care of by the loader, not by the C startup code. The C startup code only needs to know where the initial heap is located within the address space.

Let's complicate the C design further. We just stated that the C startup code only needs to know where the initial heap is located within the address space. Now, the C library is passed the full filename of the program and the command line string by the loader for use by the C global variables argc and argv[]. Figure 19-4 depicts what information is present in the address space after the C startup code is transferred control.

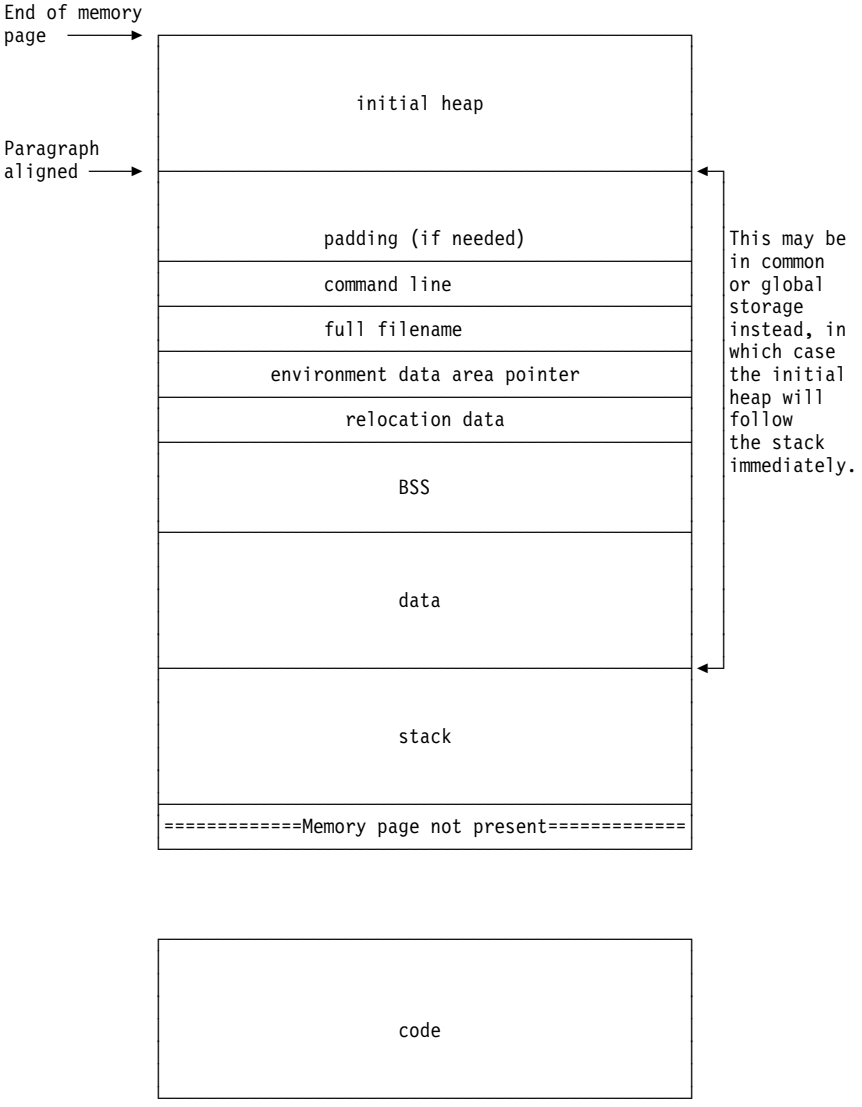


Figure 19-4. Components of a C program

The areas illustarted in Figure 19-4 are not drawn to scale, and depend on the C program being executed. The address space allocated to the stack could be much larger than the initial heap size or the address space allocated to data could be much larger than the code. Let's define some of the areas pictured in Figure 19-4

initial heap The initial heap added by the loader. The loader does not have to start the C program out with an initial heap. The C startup code can create heap space ANYWHERE in the address space. It is important to point out that heap space is not necessarily contiguous. The C heap manager will keep track of the objects devoted to the heap.

padding The padding here is to ensure that the initial heap is located on a paragraph boundary for performance reasons.

command line A list of parameters that were listed after the program name upon execution. These parameters will be parsed and inserted as values in the argv[] array.

full filename The filename of the program executing, including the fully qualified path name. If the program name run is an alias to another program, the second mentioned program (that is, not the alias) will be inserted as the filename.

environment data area pointer A pointer to the environment data area of the program. The environment data area consists of 32 bytes of information. See the "Register Interface to the CSTART module" section on page 19-13 for a more detailed description of this area.

relocation data The relocation info consists of a 32-bit count, containing the number of “sections” present in the executable program; currently, this will be either 2 or 3. This is followed by 1 32-bit field for each section, containing the virtual address offset of the start of that section in memory. These are in the order text (i.e. code), intialised data, and “BSS” (or un-initialised data). If the count word holds the value 2, there was no BSS section for this program, and that address value is omitted. These addresses can be used by a debugger to translate addresses to symbol names or to source file line numbers. A pointer to this relocation info is held in the TCB for the task. However, the C library itself does not use this information.

data The C program data area.

stack The address space devoted to contain the stack area of the C program. The system detects when the stack has grown beyond its designated space when a stack push is attempted on the page frame forced not present.

Structure of the Heap

The heap's usefulness is for providing free memory space for *malloc()*, *calloc()*, and *realloc()* C functions. The principal heap data structure will be defined as illustrated in Figure 19-5.

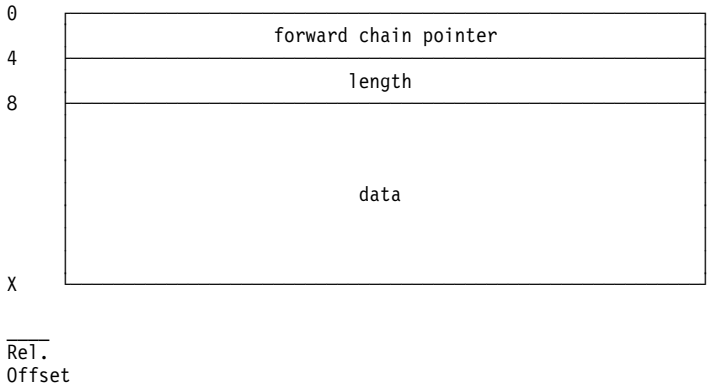


Figure 19-5. Heap data structure

Each allocation of memory from the heap will produce a structure shown in Figure 19-5. From now on, we will call an area of address space that is allocated by the heap a heap object. The heap object will be referenced using the structure above. The forward chain pointer will link the new heap object into a singly linked list of other allocated heap objects in ascending memory address order. The length field of a heap object specifies the length of the data field in bytes. The data field is the area whose address is returned by a *malloc()*, *calloc()* or *realloc()* call - it is the area to be used for inserting data. The loader can provide the C startup routine with a heap area or not. If the loader decides not to provide the C program with an initial heap area, then the heap offset passed to the C startup routine in the environment data area (described later) will be zero. See Figure 19-12 on page 19-13 for more information on the register interface to a C program.

In the case of a non-existent initial heap area, the C heap manager will have to make a call to the CP/Q Memory Manager on the first *malloc()* or *calloc()* call from the C program in order to obtain address space for a heap. The heap manager will always ask for address space to be devoted to the heap in 32K chunks or more for performance reasons. An honored request for a 32K block of address space for use by the heap manager will be stored in the heap address space as shown in Figure 19-6 on page 19-6.

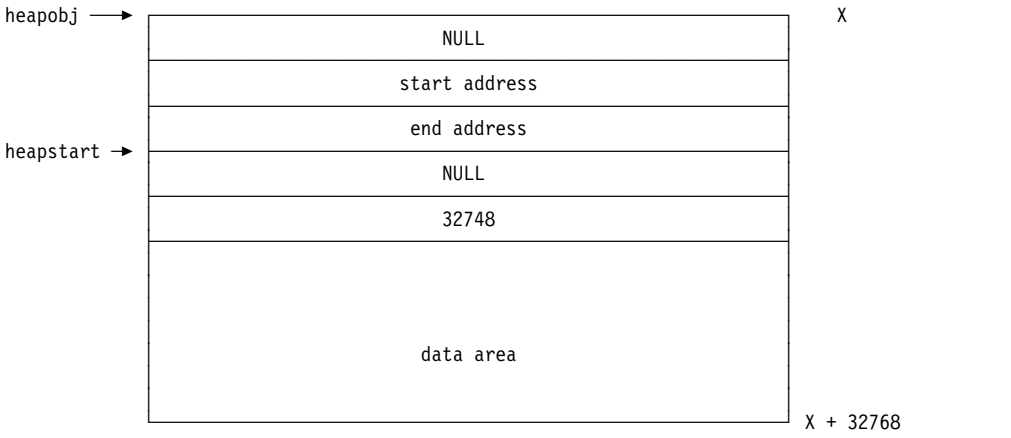


Figure 19-6. Heap space after 32K block of address space allocated

Notice in Figure 19-6 that two different structures have been created, and both structures are pointed to by different pointers. The first structure, which is pointed to by *heapobj* is called a heap object header which contains the following information in the given order:

- A 32 bit field that is used as a forward pointer.
- A 32 bit field containing the starting address of the allocated block of memory.
- A 32 bit field containing the ending address of the allocated memory.

This heap object header is not a heap object as we defined previously in Figure 19-5 on page 19-5. Rather, it is a 12 byte header that is present on each block of memory allocated on a request to the Memory Manager. This heap block header is created by the C Runtime Library heap manager and is used for purposes of not recombining heap objects across Memory Manager object boundaries during a *free()* call.

heapobj points to first heap block header. If no heap memory has been allocated, then *heapobj* equals NULL. The heap block headers are connected together as a singly linked list using the forward pointer field of the heap block header.

The *starting address* field is filled in with the value *X* which is defined in the 'Address' list to the right of the heap block header. The *ending address* field of the heap block header is filled in with the value *X + 32767*.

The second structure shown in Figure 19-6 is the actual heap object containing the forward pointer field, the length field, and the data area field. In this example let us assume that the caller asked for 32748 bytes of heap space. The heap manager found no heap objects on its 'free' chain (pointed to by *heapfree*). Therefore, the heap manager called the Memory Manager for 32K of memory. The request for 32K of memory was honored by the Memory Manager and given to the heap manager. The heap manager creates the heap block header and chains this block on the *heapobj* chain that was discussed above. The heap manager then creates a heap object with the remaining memory. The remaining memory yields a 32748 heap object due to the 12 byte overhead of the heap block header and the 8 byte overhead of the heap object pointers. Since 32748 bytes is exactly the amount requested by the caller, this heap object is chained on the heap 'inuse' chain that is

pointed to by the *heapstart* pointer. The starting address of the heap object data area is returned to the caller.

Let us now assume that a 36K block of address space has just been allocated to the heap manager due to a malloc request of 35000 bytes. The heap manager will normally request memory from the Memory Manager in units of 32K. However, this request is larger than 32K and must be rounded to the next page boundary. The heap manager will break up this 36K block of memory into three pieces: the first is the heap block header created by the heap management code, the second is the memory requested by the caller and the last piece is the remainder which goes on the heap free chain. The heap manager will break up the 36K block into the three parts as illustrated in Figure 19-7.

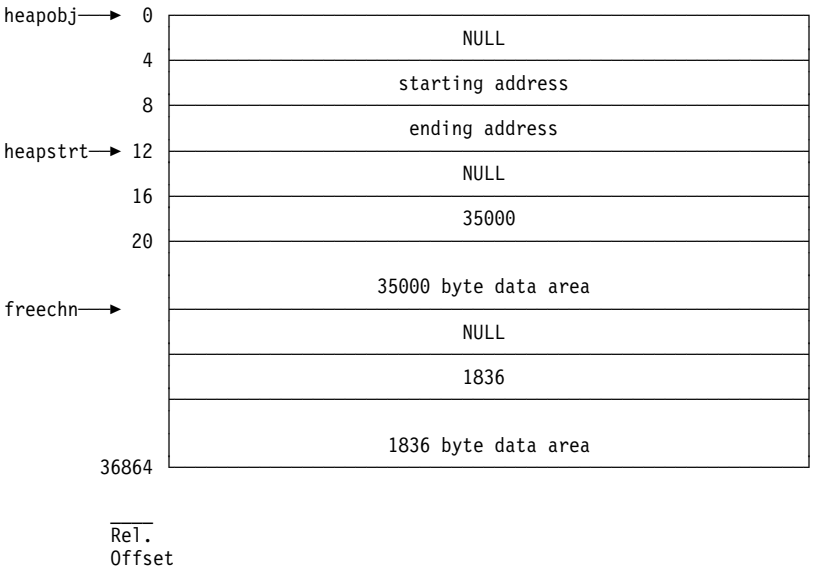


Figure 19-7. Heap space after request of 35000 bytes

To explain the structure in Figure 19-7 further, the heap block header is contained at the beginning of the 36K chunk of memory and is pointed to by *heapobj*. Following the heap block header is the 35000 byte heap object requested by the caller which is pointed to by *heapstart* (any heap requests are rounded up to the next 32 bit quantity so that each data area returned by *malloc()* or *calloc()* is aligned on a 32 bit boundary), and the remaining block of address space is kept as a heap object on the heap manager free chain for future requests. More requests to the heap manager will fragment the address space pointed to by the *freechn* pointer even further, except if that request is larger than any heap object linked on the free chain. Suppose that a request is made to the heap manager for 12K bytes of address space. As shown above, there is no heap object on the free chain that is large enough to accomodate the 12K request. The heap manager must make a request to the Memory Manager for a 32K block of address space. Assuming the request to the heap is accepted, the heap objects would be stored in the heap address space as shown in Figure 19-8 on page 19-8.

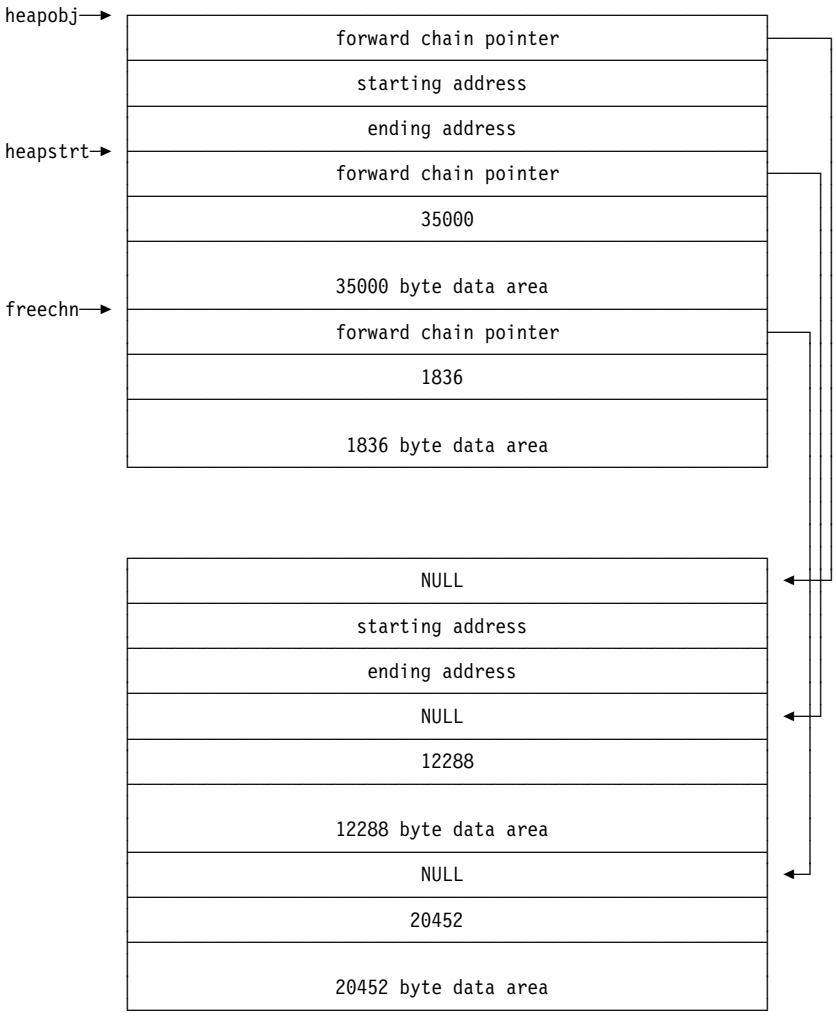


Figure 19-8. Heap space after request of 12K bytes

The heap manager created a 32K block of address space in Figure 19-8 not necessarily contiguous to any other block of heap address space. The remainder of the requested address space is put on the free chain. Notice also that another heap object has been inserted on the *heapobj* chain due to the Memory Manager request. The heap manager will try to combine any deallocated address space (deallocation by the heap manager is performed by calling the *free()* function in C) to contiguous free heap address space, provided such recombination does not cross Memory Manager object boundaries. The heap manager will always request heap address space in the private address space of the process address space. Once there is no more address space in the private address space, the heap manager will not allocate any more heap objects until heap objects are freed.

Environment Variable Implementation

Environment variables will be copied on a per-task basis. That is, environment variables will be copied from the creator task (the creator task in this instance is the task who issues the *LDLoadmod()* call) to the task containing the newly loaded load module. There will be no sharing of environment variables (for now). The LOADER will perform the action of copying the environment variables. The LOADER will use the C Runtime Library function *copyenv()* to copy environment

variables from the calling task to the task containing the newly loaded load module. Environment variables will be created in PRIVATE address space for now - this may change in the future when sharing of environment variables is implemented.

Environment variables will use the same exact structure that is used for heap objects as shown in Figure 19-9.

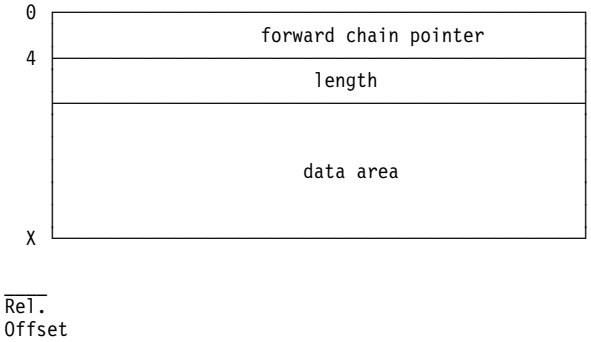


Figure 19-9. Environment object structure

The data area shown in Figure 19-9 will contain:

- An environment variable name. The variable name will be stored in upper-case.
- An equals sign ("=").
- A character string following the equals sign.

The environment will contain three singly linked lists. A head pointer (named *envstart*) will point to a singly linked list of used environment objects. Another head pointer (named *envfree*) will point to a singly linked list of free environment objects. A third head pointer (named *envobj*) will point to a singly linked list of environment objects that contain information of what address spaces have been allocated to the environment. The field *start address* in each of these environment objects contains the starting address space allocated by a Memory Manager call. The field *end address* contains the ending address of the address space allocated by a Memory Manager call. It is important for the heap to keep track of objects allocated by the Memory Manager because environment objects must not be recombined across Memory Manager object boundaries when deallocated. Note that environment objects that are chained off *envobj* do not have a length field (See Figure 19-10 on page 19-10). This length field is not necessary because if the fixed length nature of objects chained off of *envobj*.

A sample local environment could be defined in address space as shown in Figure 19-10 on page 19-10.

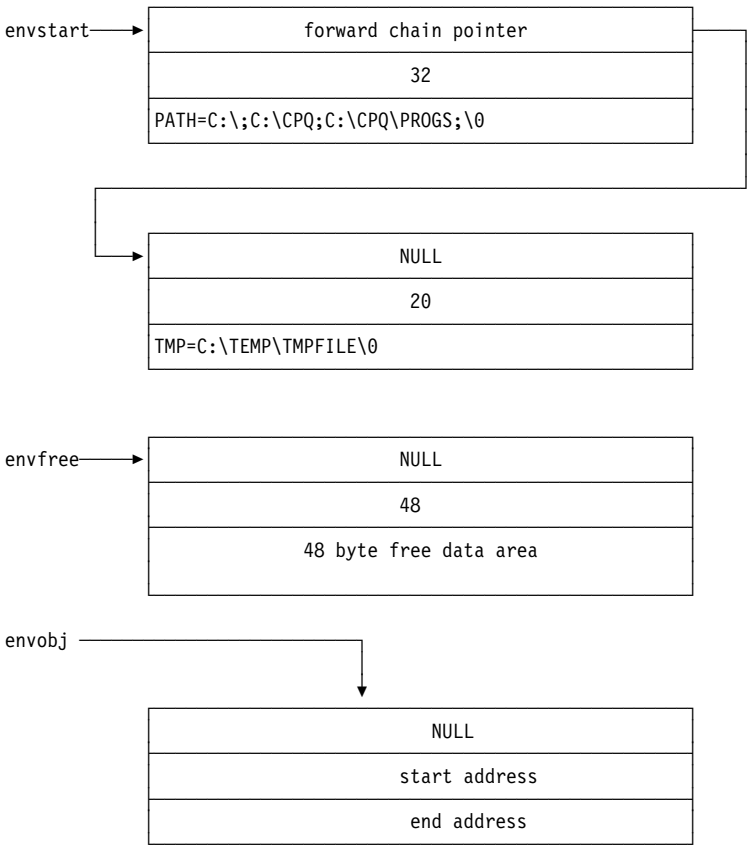


Figure 19-10. Sample local environment

In terms of the previous discussion concerning the heap, the three environment chains are analogous to the three heap chains.

Each environment string contained in the area is NULL terminated ('\0') and this NULL terminator is counted in the length field of the environment object structure.

The sample local environment has some of the exact features that are found in the heap manager:

- There is a forward chain pointer, length field (in bytes), and a data area for the environment object structure.
- All environment objects are allocated on a 32 bit boundary for performance reasons.
- New environment address space can be created by requesting free address space from the Memory Manager
- Adjacent free environment objects are automatically combined by the C Runtime Library object manager code.

Environment variables can be freed by calling the C Runtime Library *putenv()* function and specifying only the environment variable name in the parameter string. For example, the call *putenv(*"TEMP"*)* will free the environment object allocated for "TEMP" if such an environment object exists. The environment object previously containing that variable and its definition will be placed on the free chain, possibly combined with adjacent free address space in the environment area.

Environment Data Area

The environment data area is an area setup by the loader for a loaded load module as part of the C startup register interface. The environment data area consists of a 4 byte header followed by 28 bytes of environment information used by the running C program. The environment data area is illustrated in Figure 19-11.

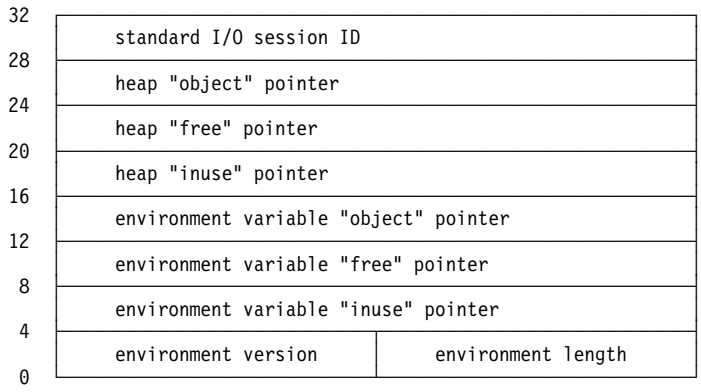


Figure 19-11. Environment Data Area

The format of the environment data area is as follows:

Header Section

environment data area version (bytes 0-1) The version number of the environment data area. Currently, the version number is set at 1.

environment data area length (bytes 2-3) The length of the environment data area, including the header, in bytes. Currently, version 1 of the environment data area is 32 (0x20) bytes.

Data Section

env. variable "inuse" pointer (bytes 4-7) The head pointer to the environment variable "inuse" chain. This pointer was referred to as *envstart* in the section "Environment Variable Implementation."

env. variable "free" pointer (bytes 8-11) The head pointer to the environment variable "free" chain. This pointer was referred to as *envfree* in the section "Environment Variable Implementation."

env. variable "object" pointer (bytes 12-15) The head pointer to the environment variable "object" chain. This pointer was referred to as *envobj* in the section "Environment Variable Implementation."

heap "inuse" pointer (bytes 16-19)	The head pointer to the heap "inuse" chain. This pointer was referred to as *heapstrt* in the section "Structure of the
heap "free" pointer (bytes 20-23)	The head pointer to the heap "free" chain. This pointer was referred to as *freechn* in the section "Structure of the Heap."
heap "object" pointer (bytes 24-27)	The head pointer to the heap "object" chain. This pointer was referred to as *heapobj* in the section "Structure of the Heap."
stdio session ID (bytes 28-31)	The session ID for streams stdin, stdout and stderr, assuming these streams have not been redirected to files. If this field contains zero, no standard I/O session ID has been created yet.

Register Interface to the CSTART Module

The CSTART module contains initialization code to create the C environment for the *main()* function. Such things as command line parsing and opening standard I/O buffers are performed by the C initialization code. The entry point for a normal C program is contained in CSTART and is called "startup".

The CSTART module needs to know about certain C program components that have been set up by the loader. The following components include: environment data area, full filename offset, command line offset, and the initial heap offset.

The CSTART module assumes that these components are set up in specific registers before CSTART is executed. The register interface to CSTART is shown in Figure 19-12.

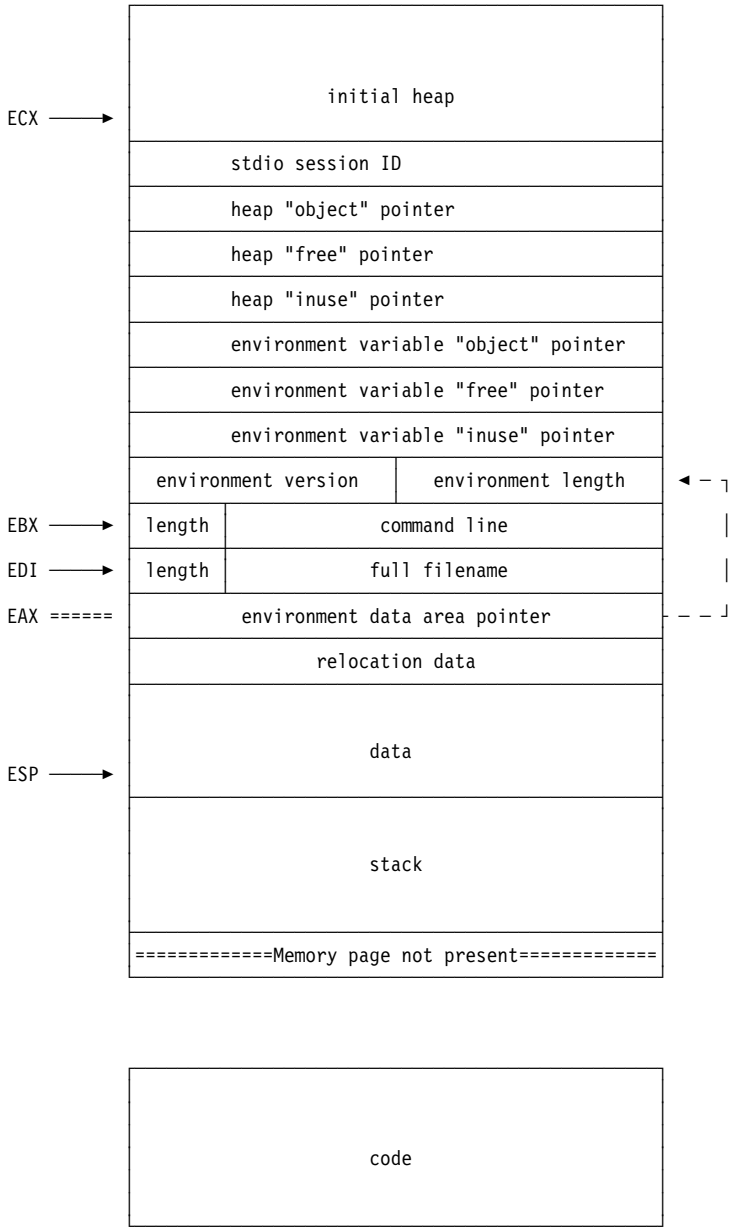


Figure 19-12. Register Interface to CSTART module

EAX Contains a pointer value which points to the environment data area if it is allocated. The environment data area consists of a 4 byte header followed by 28 bytes of environment information for use by the C program.

The format of the environment data area is as follows:

Header Area

environment data area version (bytes 0-1) The version number of the environment data area. Currently, the version number is set at 1.

environment data area length (bytes 2-3) The length of the environment data area, including the header, in bytes. Currently, version 1 of the environment data area is 32 (0x20) bytes.

Data Area

env. variable "inuse" pointer (bytes 4-7) The head pointer to the environment variable "inuse" chain. This pointer was referred to as *envstart* in the section "Structure of Environment Variables."

env. variable "free" pointer (bytes 8-11) The head pointer to the environment variable "free" chain. This pointer was referred to as *envfree* in the section "Structure of Environment Variables."

env. variable "object" pointer (bytes 12-15) The head pointer to the environment variable "object" chain. This pointer was referred to as *envobj* in the section "Structure of Environment Variables."

heap "inuse" pointer (bytes 16-19) The head pointer to the heap "inuse" chain. This pointer was referred to as *heapstrt* in the section "Structure of the Heap."

heap "free" pointer (bytes 20-23) The head pointer to the heap "free" chain. This pointer was referred to as *freechn* in the section "Structure of the Heap."

heap "object" pointer (bytes 24-27) The head pointer to the heap "object" chain. This pointer was referred to as *heapobj* in the section "Structure of the Heap."

stdio session ID (bytes 28-31) The session ID for streams stdin, stdout and stderr, assuming these streams have not been redirected to files.

If EAX is zero, then no environment data area has been allocated and the task does not have any environment variables, nor a stdio session ID defined. The CSTART module will create the C environment data area on the stack and have the environment area pointer field illustrated on Figure 19-12 on page 19-13 point to it if EAX is zero.

If the LOADER receives a non-zero value back from the *envstart* parameter in the *copyenv()* function, then the LOADER must create an environment data area in private address space and have the environment area pointer field point to it. The *copyenv()* function returns the appropriate values to insert into the environment data area. The loader may also create an environment data area in every case - if there are no environment variables defined, the LOADER must set the 3 fields in the environment data area equal to zero. The location of the environment data area (if created) may be anywhere in private address space, a suggested location would be after the command line parameters and before the initial heap, but no task (ie: a debugger) can depend on the location of the environment data area because the C Runtime Library function *putenv* may create this area. In any case the environment area pointer field illustrated on Figure 19-12 on page 19-13 will ALWAYS point to the environment data area if the environment area pointer field contains a non- null value.

EBX Contains the offset to the length field of the command line component. If EBX is zero or the length field is zero, then there is no command line.

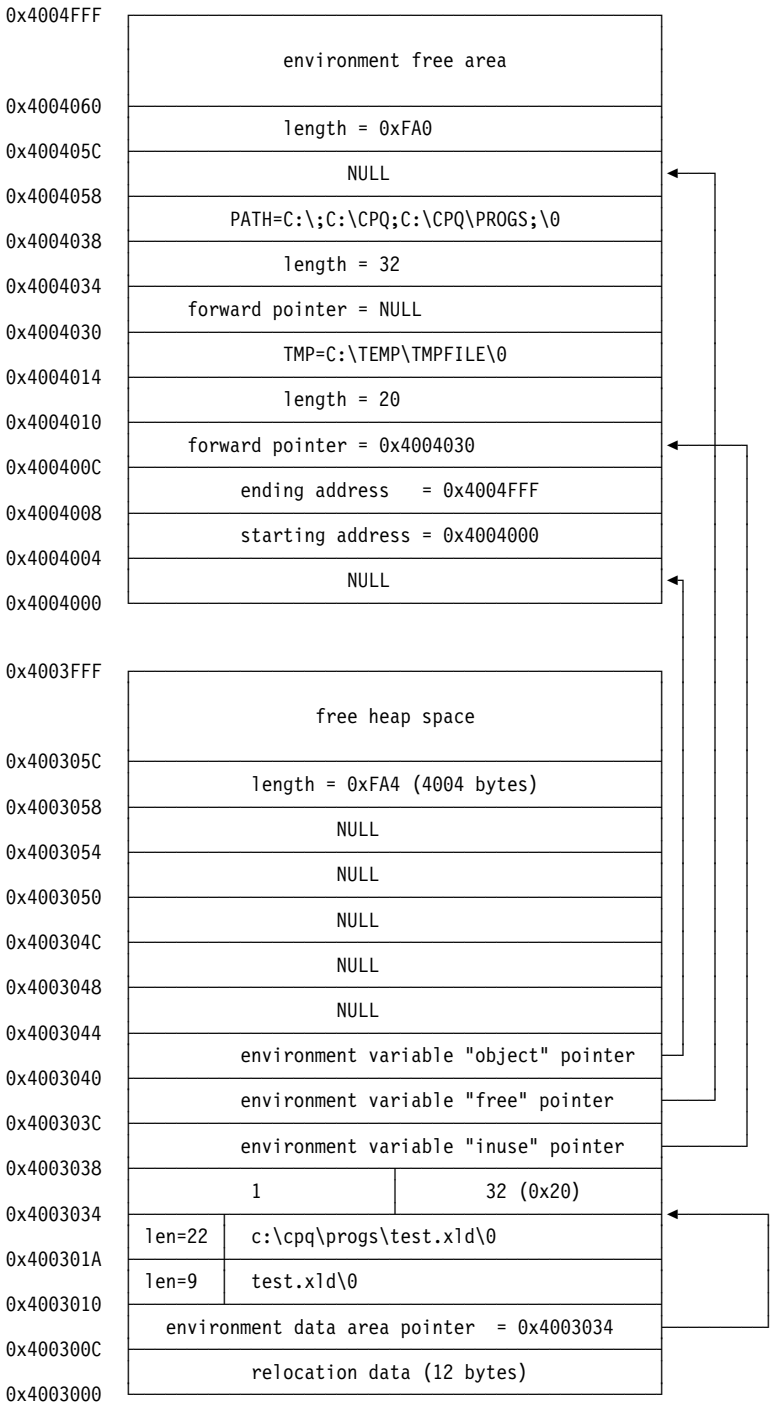
ECX Contains a pointer to the initial free heap. The initial free heap will be aligned on a 16-byte (paragraph) boundary. If ECX is zero, then there is no initial free heap allocated. NOTE: Even if there is no initial free heap, heap space can still be allocated by *malloc* calls.

EDI Contains the offset to the length field of the fully qualified file name. If EDI is zero, then there exists no fully qualified file name.

ESP Pointer to the stack. The module CSTART does not allocate stack space. Stack space must be allocated by the loader. When the system is built using SLEEP, a supervisor task will have ESP pointing partway down the stack because the stack will contain an SVC handler stack frame, which holds the "real" registers of the task.

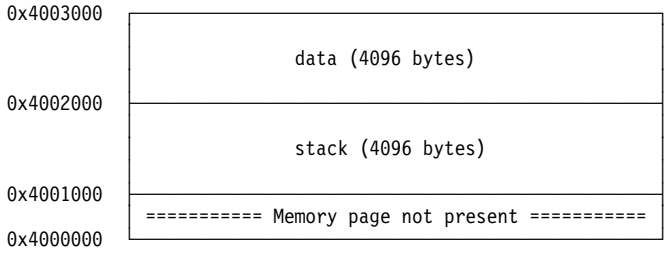
NOTE: The environment data area is optional and does not need to be allocated if there are no environment variables to be defined, no heap space needed, and no stdio streams to be opened for a task. Furthermore, the environment data area does not necessarily have to be allocated after the command line component as shown in Figure 19-12 on page 19-13. In fact, the environment data area can occur anywhere in private address space. However, the environment data area pointer field illustrated in Figure 19-12 on page 19-13 must point to the beginning of the environment data area. The environment data area is shown in Figure 19-12 on page 19-13 for purposes of illustration.

Example: Figure 19-13 on page 19-16 is an example of how the loader may set up the necessary data objects and required register interface for loaded load modules:



The points of interest concerning the data structures created in Figure 19-13 on page 19-16 are as follows:

1. The data objects in Figure 19-13 are allocated in private memory. There exist two data objects in the example:
   - Location = 0x4000000(hex) Size = 0x4000(hex) - startup data
   - Location = 0x4004000(hex) Size = 0x1000(hex) - environment variable data



NOTE: Figure not drawn to scale.

Figure 19-13 (Part 2 of 2). Example of data objects allocated by the loader

2. The memory page forced "not present" is located at 0x4000000(hex).
3. The stack has a length of 4096 bytes and is located at 0x4001000 (hex).
4. The data object of the program is exactly 4096 bytes (for simplicity) and is located at 0x4002000(hex).
5. The relocation data is this example has a length of 12 bytes.
6. The environment data area pointer field contains a pointer value that points to the environment data area located at 0x40003034(hex).
7. The full filename and command line fields are specified in order to conform with the C startup register interface.
8. The environment data area is located at 0x4003034(hex) with a header version of 1 and a length of 32 bytes.
9. The four fields after the environment variable head pointers in the environment data area are set to zero. The CSTART module will take the initial heap pointed to by ECX and set up the appropriate heap head pointers in the environment data area. The stdio session field within the environment data area is set to zero so that a session is created the first time a stdio stream is used.
10. The initial free heap follows the environment data area.
11. The second data object contains the environment variable area. The first 12 bytes are used for the environment "object" chain.
12. Two environment objects follow. These two environment objects define the environment variables "PATH" and "TMP".
13. Free environment variable area space follows the two environment objects.
14. The register values for Figure 19-13 on page 19-16 would be as follows:

EAX	0x4003034
EBX	0x400301A
ECX	0x4003054
EDX	0
ESI	0
EDI	0x4003010

Figure 19-14. Register values

Registers EDX and ESI are specifically zero because future enhancements may make use of these registers.

Appendix A. Function Summary

CP/Q C Runtime Library Function Summary

Table A-1 (Page 1 of 5). Summary of C Runtime Library Functions

Function	Prototype	Page
void abort(void)	stdlib.h	10-1
int abs(int *val*)	stdlib.h	7-1
char *asctime(const struct tm**time*)	time.h	14-1
void assert(int *val*)	assert.h	18-1
int atexit(void *(*function)(*void*)*)	stdlib.h	10-2
double atof(const char **string*)	stdlib.h	15-1
int atoi(const char **string*)	stdlib.h	15-2
long int atol(const char **string*)	stdlib.h	15-3
bcopy(char **string1*, char **string2*, int *len*)	none	13-1
int bcmp(char **string1*, char **string2*, int *len*)	none	13-2
int binsort(void **base*, size_t *nel*, size_t *elsize*, size_t *nbin*, int *(*convert_bin)(const void *a, int b), int iterations)*	stdlib.h	12-1
void brkpt(int *eax*, int *ebx*, int *ecx*, int *edx*, int *esi*, int *edi*)	stdlib.h	10-3
void *bsearch(const void **key*, const void **array*, size_t *nel*, size_t *elsize*, int (*comp)(const void **a*, const char **b*))	stdlib.h	18-2
bzero(char **string*, int *len*)	none	13-3
void *calloc(size_t *no*, size_t *size*)	stdlib.h	3-1
void clearerr(FILE **stream*)	stdio.h	5-1
clock_t clock()	time.h	14-2
int copyenv(UINT32 *totaskSVid*, UINT32 **envstart*, UINT32 **envfree*, UINT32 **envobj*, UINT32 **mem_ptr*, UINT32 *fromtaskSVid*, int *mode*)	stdlib.h	4-1
char *ctime(const time_t **tmval*)	time.h	14-3
time_t difftime(time_t *time1*, time_t *time2*)	time.h	14-4
div_t div(int *num*, int *denom*)	stdlib.h	7-2
void exit(int *rc*)	stdlib.h	10-4
void _exit(int *rc*)	stdlib.h	10-10
int fclose(FILE **stream*)	stdio.h	5-2
int feof(FILE **stream*)	stdio.h	5-3
int ferror (FILE **stream*)	stdio.h	5-4
int fflush(FILE **stream*)	stdio.h	5-5
int ffs(int i)	none	13-4
int fgetc(FILE **stream*)	stdio.h	5-6
int fgetpos(FILE **stream*, fpos_t **offptr*)	stdio.h	5-7
char *fgets(char **string*, int *n*, FILE **stream*)	stdio.h	5-8
int fileno(FILE **stream*)	stdio.h	5-9
FILE *fopen(const char **filename*, const char **mode*)	stdio.h	5-10
int fprintf(FILE **stream*, const char **format*, ...)	stdio.h	9-1
int fputc(int *c*, FILE **stream*)	stdio.h	5-12

Table A-1 (Page 2 of 5). Summary of C Runtime Library Functions

Function	Prototype	Page
int fputs(const char *_string_, FILE *_stream_)	stdio.h	5-13
int fread(void *_ptr_, size_t _ptrsize_, size_t _nitems_, FILE *_stream_)	stdio.h	5-14
void free(void *_ptr_)	stdlib.h	3-2
FILE *freopen(const char *_filename_, const char *_mode_, FILE *_stream_)	stdio.h	5-15
int fscanf(FILE *_stream_, const char *_format_, ...)	stdio.h	11-1
int fseek(FILE *_stream_, long _offset_, int _refpt_)	stdio.h	5-16
int fsetpos(FILE *_stream_, const fpos_t *_offptr_)	stdio.h	5-17
long ftell(FILE *_stream_)	stdio.h	5-18
int fwrite(const void *_ptr_, size_t _ptrsize_, size_t _nitems_, FILE *_stream_)	stdio.h	5-19
int getc(FILE *_stream_)	stdio.h	5-20
int getchar(void)	stdio.h	5-21
char *getenv(const char *_envvar_)	stdlib.h	4-3
int getenvall(char **_envbufptr_)	stdlib.h	4-4
int getenvall2(char *_envbufptr_, int envbuflen)	stdlib.h	4-5
int getopt(int _argc_, char **_argv_, char *_optstring_)		18-3
char *gets(char *_string_)	stdio.h	5-22
unsigned int getsessid()	stdio.h	9-3
struct tm *gmtime(const time_t *_tmval_)	time.h	14-5
void hsort(void *_arrayptr_, size_t _nel_, size_t _elsize_, int (*comp)(const void *_a_, const void *_b_))	stdlib.h	12-4
char *index(const char *_string1_, int _c_)	string.h	13-5
void inssort(void *_arrayptr_, size_t _nel_, size_t _elsize_, int (*comp)(const void *_a_, const void *_b_))	stdlib.h	12-6
int isalnum(int _c_)	ctype.h	16-1
int _isalnum(int _c_)	ctype.h	16-1
int isalpha(int _c_)	ctype.h	16-2
int _isalpha(int _c_)	ctype.h	16-2
int isascii(int _c_)	ctype.h	16-3
int _isascii(int _c_)	ctype.h	16-3
int iscntrl(int _c_)	ctype.h	16-4
int _iscntrl(int _c_)	ctype.h	16-4
int isdigit(int _c_)	ctype.h	16-5
int _isdigit(int _c_)	ctype.h	16-5
int isgraph(int _c_)	ctype.h	16-6
int _isgraph(int _c_)	ctype.h	16-6
int islower(int _c_)	ctype.h	16-7
int _islower(int _c_)	ctype.h	16-7
int isprint(int _c_)	ctype.h	16-8
int _isprint(int _c_)	ctype.h	16-8
int ispunct(int _c_)	ctype.h	16-9
int _ispunct(int _c_)	ctype.h	16-9
int isspace(int _c_)	ctype.h	16-10
int _isspace(int _c_)	ctype.h	16-10
int issupv(int _c_)	ctype.h	16-11
int isupper(int _c_)	ctype.h	16-12

Table A-1 (Page 3 of 5). Summary of C Runtime Library Functions

Function	Prototype	Page
int _isupper(int *c*)	ctype.h	16-12
int isxdigit(int *c*)	ctype.h	16-13
int _isxdigit(int *c*)	ctype.h	16-13
char *itoa(int *num*, char **string*, int *radix*)	stdlib.h	15-4
struct tm *localtime(const time_t **tmval*)	time.h	14-6
int labs(long *lval*)	stdlib.h	7-3
ldiv_t ldiv(long *num*, long *denom*)	stdlib.h	7-4
void longjmp(jmp_buf *env*, int *retval*)	setjmp.h	10-5
void *malloc(size_t *size*)	stdlib.h	3-3
char *memccpy(void **buffer1*, const void **buffer2*, int *c*, size_t *cnt*)	string.h	8-1
void *memchr(const void **buffer*, int *c*, size_t *cnt*)	string.h	8-2
int memcmp(const void **buffer1*, const void **buffer2*, size_t *cnt*)	string.h	8-3
void *memcpy(void **buffer1*, const void **buffer2*, size_t *cnt*)	string.h	8-4
int memicmp(const void **buffer1*, const void **buffer2*, size_t *cnt*)	string.h	8-5
void *memmove(void **buffer1*, const void **buffer2*, size_t *cnt*)	string.h	8-6
void *memset(void **buffer*, int *c*, size_t *cnt*)	string.h	8-7
time_t mktime(struct tm **time*)	time.h	14-7
void msort(void **arrayptr*, size_t *nel*, size_t *elsize*, int (*comp)(const void **a*, const void **b*))	stdlib.h	12-7
void paws(int *status*)	stdlib.h	10-6
void perror(const char **string*)	stdio.h	18-5
int printf(const char **format*, ...)	stdio.h	9-4
int putc(int *c*, FILE **stream*)	stdio.h	5-23
int putchar(int *c*)	stdio.h	5-24
int putenv(char **envstring*)	stdlib.h	4-6
int puts(const char **string*)	stdio.h	5-25
void qsort(void **arrayptr*, size_t *nel*, size_t *elsize*, int (*comp)(const void **a*, const void **b*))	stdlib.h	12-8
int raise(int **sig*)	signal.h	18-6
int rand()	stdlib.h	18-7
void *realloc(void **oldptr*, size_t *size*)	stdlib.h	3-4
int remove(const char **filename*)	stdio.h	5-26
int rename(const char **oldfilename*, const char **newfilename*)	stdio.h	5-27
char *rindex(const char **string1*, int *c*)	string.h	13-6
int scanf(const char **format*, ...)	stdio.h	11-3
void setattr(char *color*)	stdio.h	9-5
void setbuf(FILE **stream*, char **string*)	stdio.h	5-29
int setjmp(jmp_buf *env*)	setjmp.h	10-7
unsigned int setsessid(unsigned int *newsessid*)	stdio.h	9-6
int setvbuf(FILE **stream*, char **string*, int *type*, size_t *size*)	stdio.h	5-30
void (* signal(int *sig*, void *(* function)(*int*)))(int)	signal.h	18-8
int sprintf(char **string*, const char **format*, ...)	stdio.h	9-7
void srand(unsigned int *seed*)	stdlib.h	18-10
int sscanf(const char **string*, const char **format*, ...)	stdio.h	11-4
char *strcat(char **string1*, const char **string2*)	string.h	13-7

Table A-1 (Page 4 of 5). Summary of C Runtime Library Functions

Function	Prototype	Page
int strchr(const char *_string_, int _c_)	string.h	13-8
int strcmp(const char *_string1_, const char *_string2_)	string.h	13-9
char *strcpy(char *_string1_, const char *_string2_)	string.h	13-10
size_t strcspn(const char *_string1_, const char *_string2_)	string.h	13-11
char *strdup(char *_string1_)	string.h	13-12
char *strerror(int _errnum_)	string.h	13-13
size_t strftime(char *_buffer_, size_t _bufmax_, const char *_format_, const struct tm *_time_)	time.h	14-8
int stricmp(const char *_string1_, const char *_string2_)	string.h	13-14
size_t strlen(const char *_string_)	string.h	13-15
char *strlwr(char *_string_)	string.h	13-16
char *strncat(char *_string1_, const char *_string2_, size_t _n_)	string.h	13-17
int strncmp(const char *_string1_, const char *_string2_, size_t _n_)	string.h	13-18
int strnicmp(const char *_string1_, const char *_string2_, size_t _n_)	string.h	13-19
char *strncpy(char *_string1_, const char *_string2_, size_t _n_)	string.h	13-20
char *strpbrk(const char *_string1_, const char *_string2_)	string.h	13-21
char *strrchr(const char *_string1_, int _c_)	string.h	13-22
char *strrev(char *_string_)	string.h	13-23
size_t strspn(const char *_string1_, const char *_string2_)	string.h	13-24
char *strstr(const char *_string1_, const char *_string2_)	string.h	13-25
char *strtok(char *_string1_, const char *_string2_)	string.h	13-26
long int strtol(const char *_string_, char **_endofstr_, int _base_)	stdlib.h	13-27
unsigned long int strtoul(const char *_string_, char **_endofstr_, int _base_)	stdlib.h	13-28
char *strupr(char *_string_)	string.h	13-29
int swab(char *_buffer1_, char *_buffer2_, int _cnt_)		8-8
int system(const char *_cmdstring_)	stdlib.h	10-8
int tolower(int _c_)	ctype.h	15-5
int _tolower(int _c_)	ctype.h	15-5
int toupper(int _c_)	ctype.h	15-6
int _toupper(int _c_)	ctype.h	15-6
int rewind(FILE *_stream_)	stdio.h	5-28
time_t time(time_t *_tmval_)	time.h	14-10
FILE *tmpfile()	stdio.h	5-31
char *tmpnam(char *_s_)	stdio.h	5-32
int ungetc(int _c_, FILE *_stream_)	stdio.h	5-33
type va_arg(va_list _ap_, _type_)	stdarg.h	17-1
void va_end(va_list _ap_)	stdarg.h	17-2
void va_start(va_list _ap_, _parmN_)	stdarg.h	17-3
int vfprintf(FILE *_stream_, const char *_format_, va_list _arglist_)	stdio.h	9-8
int vprintf(const char *_format_, va_list _arglist_)	stdio.h	9-9
int vsprintf(char *_string_, const char *_format_, va_list _arglist_)	stdio.h	9-10

Appendix B. Errno codes

Below is a table that describes the possible error codes that can be returned by the C Runtime Library global variable *errno*. The *errno* symbols and values listed below are defined in the *OSF/1 POSIX Conformance Document*. These codes can also be found in the C Runtime Library include file *errno.h*.

Table B-1 (Page 1 of 4). Possible return codes for ERRNO

Name	Value	Description
ESUCCESS	0	Success
EPERM	1	Operation not permitted
ENOENT	2	No such file or directory
ESRCH	3	No such process or directory
EINTR	4	Interrupted function call
EIO	5	I/O error
ENXIO	6	No such device or address
E2BIG	7	Argument list too long
ENOEXEC	8	Exec format error
EBADF	9	Bad file descriptor
ECHILD	10	No child process
EAGAIN	11	Resource deadlock avoided
ENOMEM	12	Not enough space
EACCES	13	Access denied
EFAULT	14	Bad address
ENOTBLK	15	Block device required
EBUSY	16	Resource busy
EEXIST	17	File exists
EXDEV	18	Improper link
ENODEV	19	No such device
ENOTDIR	20	No a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	Too many files open in system
EMFILE	24	Too many open files
ENOTTY	25	Inappropriate I/O control operation
ETXTBSY	26	Text file busy
EFBIG	27	File too large
ENOSPC	28	No space left no device
ESPIPE	29	Invalid seek
EROFS	30	Read-only file system
EMLINK	31	Too many open files
EPIPE	32	Broken pipe
EDOM	33	Domain error
ERANGE	34	Result too large
EAGAIN	35	Resource temporarily unavailable

Table B-1 (Page 2 of 4). Possible return codes for ERRNO

Name	Value	Description
EWOULDBLOCK	35	Operation would block
EINPROGRESS	36	Operation now in progress
EALREADY	37	Operation already in progress
ENOTSOCK	38	Socket operation on non-socket
EDESTADDRREQ	39	Destination address required
EMSGSIZE	40	Message too long
EPROTOTYPE	41	Protocol wrong type for socket
ENOPROTOOPT	42	Protocol not available
EPROTONOSUPPORT	43	Protocol not supported
ESOCKTNOSUPPORT	44	Socket type not supported
EOPNOTSUPP	45	Operation not supported on socket
EPFNOTSUPPORT	46	Protocol family not supported
EAFNOTSUPPORT	47	Address family not supported by protocol family
EADDRINUSE	48	Address already in use
EADDRNOTAVAIL	49	Can't assign requested address
ENETDOWN	50	Network is down
ENETUNREACH	51	Network is unreachable
ENETRESET	52	Network dropped connection on reset
ECONNABORTED	53	Software caused connection abort
ECONNRESET	54	Connection reset by peer
ENOBUFS	55	No buffer space available
EISCONN	56	Socket already in use
ENOTCONN	57	Socket is not connected
ESHUTDOWN	58	Can't send after socket shutdown
ETOOMANYREFS	59	Too many references: can't splice
ETIMEDOUT	60	Connection timed out
ECONNREFUSED	61	Connection refused
ELOOP	62	Too many levels of symbolic links
ENAMETOOLONG	63	File name too long
EHOSTDOWN	64	Host is down
EHOSTUNREACH	65	No route to host
ENOTEMPTY	66	Directory not empty
EPROCLIM	67	Too many processes
EUSERS	68	Too many users
EDQUOT	69	Disk quota exceeded
ESTALE	70	Stale file system
EREMOTE	71	Too many levels of remote in path
EBADRPC	72	RPC struct is bad
ERPCMISMATCH	73	RPC version wrong
EPROGUNAVAIL	74	RPC program not available
EPROGMISMATCH	75	Program version wrong
EPROGUNAVAIL	76	Bad procedure for program
ENOLCK	77	No locks available

Table B-1 (Page 3 of 4). Possible return codes for ERRNO

Name	Value	Description
ENOSYS	78	Function not implemented
ENOMSG	80	No message of desired type
EIDRM	81	Identifier removed
ENOSR	82	Out of streams resources
ETIME	83	System call timed out
EBADMSG	84	Next message has wrong type
EPROTO	85	Streams protocol error
ENODATA	86	No message on stream head read queue
ENOSTR	87	File descriptor not associated with a stream
ECLONEME	88	Tells *open()* to clone the device
EDIRTY	89	Mounting a dirty filesystem without force
EDUPPKG	90	Duplicate package name on install
EVERSION	91	Version number mismatch
ENOPKG	92	Unresolved package name
ENOSYM	93	Unresolved symbol name
ESOFT	123	Correctable disk error
EMEDIA	124	Hard ECC or similar disk media failure

Appendix C. Sample Link Control File

The following is an example of a linker control file for the *BIND* program. *BIND* requires extended memory under DOS because it runs as a protect mode program under Phar Lap's DOS/Extender.

The CLIB.LIB archive module contains the C startup modules CSTART.XCF, CINIT.XCF, and EXIT.XCF. If you wish to have your own startup modules inserted instead, then you must specify an entry point other than *startup* because that is the name of the function within the CSTART.XCF module.

If you wish to use the reduced code size startup modules, such as CINITNON.XCF and/or EXITNON.XCF, then you must place an "insert" command line BEFORE the "insert clib.lib" command line in your BIND control file so that the desired modules get overridden within the C Library archive module.

See Figure C-1 for an illustration of a sample BIND control file.

```
* This is a sample link command file.  Comments require a * in
* column one.  You can redirect standard input from this file,
* or you can start the binder and then use the EXECute command
* to run this file.

setopt quiet
setopt wantkey
setopt verbose
setopt loadmap= sample.map

* Place your object files here:
insert sample.xcf

* Library insertions follow.

* cinitnon does not use the file support  (this
* alternate object files are supplied to permit smaller load modules
* if file is not needed; if uncommented, they will
* override cinit in the clib.lib file):
* insert cinitnon.xcf

* exitnon does not use the file support and
* should be used with cinitnon (if uncommented, it will override exit in
* clib.lib):
* insert exitnon.xcf

insert clib.lib
insert cpqlib.lib
insert conlib.lib
insert fslib.lib
insert ldlib.lib
insert smlib.lib

* Resolve external references between object files:
resolve

* Main load module entry point (note that symbol names are
* case sensitive):
entry startup

* Garbage Collect unused routines so they are not included in the
* load module:
gc

* Check for unresolved external references:
er full

* Print a map:
map

* Save the load module to a file (our convention will be that
* executable XCOFF files will use the extension .XLD to distinguish
* them from non-executables):
save ll sample.xld
```

Figure C-1. Sample Link Control File

Artwork Definitions

LERS Definitions

Table Definitions

Figures

Headings

		6-5	tell	
WRITE	NSLCDFL			
		6-6	write	
ABS	NSLCMATH			
		7-1	abs	
				A-1
DIV	NSLCMATH			
		7-2	div	
				A-1
LABS	NSLCMATH			
		7-3	labs	
				A-3
LDIV	NSLCMATH			
		7-4	ldiv	
				A-3
MEMCCPY	NSLCMEM			
		8-1	memccpy	
				A-3
MEMCHR	NSLCMEM			
		8-2	memchr	
				A-3
MEMCMP	NSLCMEM			
		8-3	memcmp	
				A-3
MEMCPY	NSLCMEM			
		8-4	memcpy	
				A-3
MEMICMP	NSLCMEM			
		8-5	memicmp	
				A-3
MEMMOVE	NSLCMEM			
		8-6	memmove	
				A-3
MEMSET	NSLCMEM			
		8-7	memset	
				A-3
SWAB	NSLCMEM			
		8-8	swab	
				A-4
FPRINTF	NSLCPRT			
		9-1	fprintf	
				A-1
GTSID	NSLCPRT			
		9-3	getsessid	
				A-2
PRINTF	NSLCPRT			
		9-4	printf	
				A-3
SETATTR	NSLCPRT			
		9-5	setattr	
				A-3
SETSID	NSLCPRT			
		9-6	setsessid	
				A-3
SPRINTF	NSLCPRT			
		9-7	sprintf	
				A-3
VFPRNTF	NSLCPRT			
		9-8	vfprintf	
				A-4
VPRINTF	NSLCPRT			
		9-9	vprintf	
				A-4
VSPRNTF	NSLCPRT			
		9-10	vsprintf	
				A-4
ABORT	NSLCXT			
		10-1	abort	
				A-1
ATEXIT	NSLCXT			
		10-2	atexit	
				A-1
BRKPT	NSLCXT			
		10-3	brkpt	
				A-1
EXIT	NSLCXT			
		10-4	exit	
				A-1
LONGJMP	NSLCXT			
		10-5	longjmp	
				A-3

Footnotes

id	File	Page	References
STABLE	NSLCSORT	12-5	1 12-5, 12-6, 12-7, 12-8

Tables

id	File	Page	References
FOPMOD	NSLCFL	5-10	5-1
SEEK	NSLCFL	5-16	5-2
SETVBUF	NSLCFL	5-30	5-3
LSEEK	NSLCDFL	6-2	6-1
OPMOD	NSLCDFL	6-3	6-2
PRTFLG	NSLCPRT	9-1	9-1
PRTTYP	NSLCPRT	9-2	9-2
SCNTYP	NSLCSCN	11-2	11-1
TMTYPE	NSLCTM	14-9	14-1
SIGNAM	NSLCMSC	18-8	18-1 18-6, 18-8
CRLTBL	NSLCTABL	A-1	A-1

Processing Options

Runtime values:

Document fileid	LSLAPL2 SCRIPT
Document type	USERDOC
Document style	IBMXAGD
Profile	EDFPRF40
Service Level	0032
SCRIPT/VS Release	4.0.0
Date	98.06.17
Time	14:29:29
Device	PSA
Number of Passes	3
Index	YES
SYSVAR G	INLINE
SYSVAR X	YES

Formatting values used:

Annotation	NO
Cross reference listing	YES
Cross reference head prefix only	NO
Dialog	LABEL
Duplex	YES
DVCF conditions file	(none)
DVCF value 1	(none)
DVCF value 2	(none)
DVCF value 3	(none)
DVCF value 4	(none)
DVCF value 5	(none)
DVCF value 6	(none)
DVCF value 7	(none)
DVCF value 8	(none)
DVCF value 9	(none)
Explode	NO
Figure list on new page	NO
Figure/table number separation	YES
Folio-by-chapter	YES
Head 0 body text	(none)
Head 1 body text	Chapter
Head 1 appendix text	Appendix
Hyphenation	NO
Justification	NO
Language	ENGL
Keyboard	395
Layout	OFF
Leader dots	YES
Master index	(none)
Partial TOC (maximum level)	4
Partial TOC (new page after)	INLINE
Print example id's	NO
Print cross reference page numbers	YES
Process value	(none)
Punctuation move characters	.,
Read cross-reference file	(none)
Running heading/footing rule	NONE
Show index entries	NO
Table of Contents (maximum level)	4
Table list on new page	YES
Title page (draft) alignment	RIGHT
Write cross-reference file	(none)

Imbed Trace

Page 0	CPQSET
Page xiii	APL2SOA
Page xvii	NSLCPRE
Page 1-1	NSLCINT
Page 2-1	NSLCGLB
Page 3-1	NSLCALL
Page 4-1	NSLCENV
Page 5-1	NSLCFL
Page 6-1	NSLCDFL
Page 7-1	NSLCMATH
Page 8-1	NSLCMEM
Page 9-1	NSLCPRT
Page 10-1	NSLCXT
Page 11-1	NSLCSCN
Page 12-1	NSLCSORT
Page 13-1	NSLCSTR
Page 14-1	NSLCTM
Page 15-1	NSLCTO
Page 16-1	NSLCIS
Page 17-1	NSLCVA
Page 18-1	NSLCMSC
Page 19-1	NSLCTECH
Page A-1	NSLCTABL
Page B-1	NSLCERNO
Page C-1	NSLCLNK