DPS 6
GCOS 6 MOD 400
C USER'S GUIDE

SUBJECT

DPS 6 C Programming Language

SPECIAL INSTRUCTIONS

This manual supersedes the *DPS 6 GCOS 6 MOD 400 C User's Guide*, Order No. CW35-01 dated September 1985. Change bars in the margins indicate changes and additions, while asterisks indicate deletions.

SOFTWARE SUPPORTED

The C compiler Release 2.0 executes running under Release 4.0 of the MOD 400 Executive.

ORDER NUMBER

CW35-02

March 1986

Honeywell

PREFACE

This manual describes the C programming language as implemented under MOD 400. The language is described by noting variations from a baseline version of C. The reader is assumed to be familiar with C. This manual is not a language specification, nor is it intended as a tutorial document.

The new C functions supported are:

getptcb	putr	runvp	ucf_defr
getr	runl	setprint	ucf_finish
gettcb	runlp	tzset	ucf_init
posr	runv	ucf_defc	

The new C-related utilities supported are:

CSICK	DL_ENV	ENV_DEF	GET_ENV
LIST_ENV	SET_ENV		

Descriptions of the SL and FILE_OUT commands have been moved to the DPS 6 GCOS 6 MOD 400 Commands manual.

Section 1 defines the version of C used as the basis for comparison.

Section 2 notes all variations in the MOD 400 implementation of the C language.

Section 3 describes the process of developing C programs under MOD 400, including use of the C compilers and loading C programs under MOD 400.

Section 4 lists the C standard library as implemented under MOD 400.

USER COMMENTS FORMS are included at the back of this manual. These forms are to be used to record any corrections, changes, or additions that will make this manual more useful.

File No.: 1R23, 1S23, 1623

CW35-02

Appendix A is a list of C compiler diagnostic messages.

Appendix B lists the eight-bit ASCII character set.

A glossary defines UNIX, C, and MOD 400 terms.

Braces { } in this manual are used to enclose information from which the user must make a choice.

The following conventions are used to indicate the relative levels of topic headings used in this manual:

Level	Format
1 (highest)	ALL CAPITAL LETTERS, UNDERLINED
2	Initial Capital Letters, Underlined
3	ALL CAPITAL LETTERS, NOT UNDERLINED
4	Initial Capital Letters, Not Underlined

MANUAL DIRECTORY

The following publications constitute the GCOS 6 MOD 400 manual set. See the "Software/Manual Matrix" of the Guide to Software Documentation for the current revision number and addenda (if any) of the manuals.

Manuals are obtained by submitting a Honeywell Publications Order Form to the following address:

Honeywell Information Systems Inc.
47 Harvard Street
Westwood, MA 02090
Att: Publications Services

Honeywell software reference manuals are periodically updated to support enhancements and improvements to the software. Before ordering any manual listed below, the customer should refer to the Guide to Software Documentation to obtain information concerning the specific edition of the manual that supports the software currently in use at the installation. When specifying manuals on the Publications Order Form, a customer using the 4-digit base publication number listed below will obtain the latest edition of the manual currently in stock. The Publications Distribution Center can provide specific editions of a publication only when supplied with the 7- or 8-character order number described in the Guide to Software Documentation.

Honeywell applications software packages - such as INFO, the Honeywell Manufacturing System (HMS), and TPS 6 - provide specialized services. See your Honeywell representative for information concerning the availability of applications software and supporting documentation.

Base Publication Number	Manual Title
CW35	GCOS 6 C User's Guide
CZ01	GCOS 6 MOD 400 Guide to Software Documentation
CZ02	GCOS 6 MOD 400 System Building and Administration
CZ03	GCOS 6 MOD 400 System Concepts
CZ04	GCOS 6 MOD 400 System User's Guide
CZ05	GCOS 6 MOD 400 System Programmer's Guide - Volume I
CZ06	GCOS 6 MOD 400 System Programmer's Guide - Volume II
CZ07	GCOS 6 MOD 400 Programmer's Pocket Guide
CZ09	GCOS 6 MOD 400 System Maintenance Facility Administrator's Guide
CZ10	GCOS 6 MOD 400 Menu System User's Guide
CZ11	GCOS 6 MOD 400 Software Installation Guide
CZ15	GCOS 6 MOD 400 Application Developer's Guide
CZ16	GCOS 6 MOD 400 System Messages
CZ17	GCOS 6 MOD 400 Commands
CZ18	GCOS 6 Sort/Merge
CZ19	GCOS 6 Data File Organizations and Formats
CZ20	GCOS 6 MOD 400 Transaction Control Language Facility
CZ21	GCOS 6 MOD 400 Display Formatting and Control
CZ22	GCOS 6 VISION Reference Manual
CZ23	DM6 AZ7 Reference Card
CZ24	Introduction to DM6 AZ7 Query Writing
CZ25	DM6 AZ7 Reference Manual
CZ29	GCOS 6 VISION Reference Card
CZ31	GCOS 6 Advanced COBOL Compiler User's Guide
CZ32	GCOS 6 Multiuser COBOL Compiler Guide
CZ34	GCOS 6 COBOL 74 Language Reference
CZ35	GCOS 6 COBOL Quick Reference Guide
CZ36	GCOS 6 BASIC Reference
CZ37	GCOS 6 BASIC Quick Reference Guide
CZ38	GCOS 6 Assembly Language (MAP) Reference
CZ39	GCOS 6 Advanced FORTRAN Reference
CZ40	GCOS 6 Pascal User's Guide
CZ42	GCOS 6 Ada Compiler System User's Guide
CZ52	DM6 I-D-S/II Programmer's Guide
CZ53	DM6 I-D-S/II Data Base Administrator's Guide
CZ54	DM6 I-D-S/II Reference Card
CZ70	Electronic Mail Facility Administrator's Guide
CZ71	DM6 TP Development Reference
CZ72	DM6 TP Application User's Guide
CZ73	DM6 TP Forms Processing
CZ74	GCOS 6 Data Base Augmented Real-Time Tracing System User's Guide
CZ93	Electronic Mail Facility User's Guide
GZ13	GCOS 6 MOD 400 Release 4.0 Migration Guide

Base Publication Number	Manual Title
HC01	MOD 400 Application Development Overview
HC12	Disk-Based Data Entry Facility-II User's Guide
HC13	Disk-Based Data Entry Facility-II Operator's Quick Reference Guide

The following manuals describe the MOD 400 distributed processing software components:

Base Publication Number	Manual Title
CB35	DPS 6/DPS 7 PVE File Transfer Facility User's Guide
CF11	DPS 6/DPS 7 PVE Remote Batch Facility User's Guide
CG90	Interactive Entry Facility-II User's Guide
CZ59	Level 6 to Level 6 File Transmission Facility User's Guide
CZ60	Level 6 to Level 66 File Transmission Facility User's Guide
CZ61	Level 6 to Level 62 File Transmission Facility User's Guide
CZ62	BSC Transport Facility User's Guide
CZ63	2780/3780 Workstation Facility User's Guide
CZ64	HASP Workstation Facility User's Guide
CZ65	Programmable Facility/3271 User's Guide
CZ66	Remote Batch Facility/66 User's Guide
GG19	Disk-Based VIP7305 Emulator Facility User's Guide
GG20	Disk-Based Asynchronous Communications Facility User's Guide
GT18	Disk-Based VIP7705 Emulator Facility User's Guide
GT19	Disk-Based VIP7814 Emulator Facility User's Guide

The following manuals describe the ORACLE data base management facility:

Base Publication Number	Manual Title
GS61	GCOS 6 MOD 400 ORACLE Installation Guide
GS62	GCOS 6 MOD 400 ORACLE Database Administrator's Guide
GS63	GCOS 6 MOD 400 ORACLE Interactive Application Facility (IAF) Terminal Operator's Guide
GS64	GCOS 6 MOD 400 ORACLE Interactive Application Facility (IAF) Terminal Operator's Reference Manual
GS65	GCOS 6 MOD 400 ORACLE Interactive Application Facility (IAF) Designer's Guide
GS66	GCOS 6 MOD 400 ORACLE Interactive Application Facility (IAF) Designer's Reference Manual
GS67	GCOS 6 MOD 400 ORACLE HLI Precompiler Interface
GS68	GCOS 6 MOD 400 ORACLE Host Language Call Interface Manual
GS69	GCOS 6 MOD 400 ORACLE RPF Report Text Formatter User's Guide
GS70	GCOS 6 MOD 400 ORACLE RPT Report Generator User's Guide
GS71	GCOS 6 MOD 400 ORACLE SQL/UFI Reference Manual
GS72	GCOS 6 MOD 400 ORACLE Terminal User's Guide
GS73	GCOS 6 MOD 400 ORACLE Utilities Manual
GS74	GCOS 6 MOD 400 ORACLE Error Messages and Codes

In addition, the following publications provide supplementary information:

Base Publication Number	Manual Title
AS22	Level 6 Models 6/34, 6/36, and 6/43 Minicomputer Handbook
AT97	Level 6 Communications Handbook
CC71	Level 6 Minicomputer Systems Handbook
CD18	Level 6 MOD 400/600 Online Test and Verification Operator's Guide
FQ41	Writable Control Store User's Guide

These five manuals are not covered by the Guide to Software Documentation. See your Honeywell representative for information concerning the versions of the manuals relevant to your installation.

Users should be aware that a software release bulletin accompanies each software product ordered from Honeywell. Users should consult the software release bulletin before using the software. Users should contact their Honeywell representative if a copy of the software release bulletin is not available.

CONTENTS

Page

CONTENTS

Page

CONTENTS

Page

Page

CONTENTS

ILLUSTRATIONS

TABLES

Section 1
INTRODUCTION

C is a general-purpose, low-level programming language. It was developed under UNIX,* but is now available for use with a number of computers and operating systems.

This manual describes the C programming language as implemented on DPS 6 systems under MOD 400. The language is described by noting variations from a baseline version of C.

The MOD 400 C compiler provides a C program with an emulation of the UNIX environment. The environment simulated is a single-user system. Run-time routines, signals, messages, and traps all appear to a C program as they do under UNIX.

The reader is assumed to be familiar with C, UNIX, and MOD 400. This manual is not a complete reference document; nor is it intended as a tutorial document.

DEFINITION OF "BASELINE" C

The version of C used in this manual as the baseline for comparison is as described in:

> The C Programming Language, by Brian W. Kernighan and Dennis M. Ritchie. 1978, Prentice-Hall, Inc., Englewood Cliffs, NJ

The phrase "baseline C" is used in this manual to refer to that version of C.

*UNIX is a Trademark of Bell Laboratories.

You are assumed to have a copy of this book on hand when you refer to this manual.

CONTENTS OF THIS MANUAL

The rest of this manual is organized as follows:

Section 2 notes all variations in the MOD 400 implementation of the C language. The section is organized to match Appendix A of The C Programming Language.

Section 3 describes the process of developing C programs under MOD 400, including use of the C compiler, various C utilities, and loading of C programs.

Section 4 lists the C standard library of run-time routines.

Appendix A lists the C compiler diagnostic messages.

Appendix B lists the eight-bit ASCII collating sequence.

A glossary defines UNIX, C, and MOD 400 terms.

Section 2
IMPLEMENTATION OF THE C LANGUAGE

This section lists variations from the baseline C as described in The C Programming Language (see Section 1). This section is organized and keyed to match Appendix A ("The C Reference Manual") of that book. Bracketed numbers in level heads appearing in this section correspond to headings in Appendix A of that book.

This section contains only statements of variations. If a feature is not described in this section, it is fully supported by the C compiler, and behaves exactly the same as in baseline C.

LEXICAL CONVENTIONS [2]

The following variations on baseline C lexical conventions exist in MOD 400 C.

Identifiers (Names) [2.2]

An identifier name can contain uppercase or lowercase letters, digits, underscores, and dollar signs, in any order. Only the first eight characters are significant. External identifiers are mapped to six characters, in uppercase. The MOD 400 C compiler treats external identifiers as follows:

1. All lowercase characters are changed to uppercase characters.

2. All underscores are removed.

3. If more than six characters remain after eliminating underscores, vowels are eliminated from right to left until either: (1) there are only six characters left, or (2) there are no more vowels.

4. If more than six characters remain after eliminating underscores and vowels, the excess is truncated, right to left.

Keywords [2.3]

The following additional identifiers are reserved for use as keywords:

```
void
enum
escape
const
```

Constants [2.4]

The following variations on baseline C constants exist in MOD 400 C.

Strings [2.5]

A string has type "array of characters" and storage class const, and is initialized with the given characters.

Hardware Characteristics [2.6]

The size of C data types are:

Data Type	Size (bits)
char	8
unsigned char	8
int	16
unsigned int	16
short	16
long	32
unsigned long	32
float	32
double	64

WHAT'S IN A NAME? [4]

The C compiler supports all arithmetic types. C data types are described below.

A character variable (char) is a one-byte, signed binary integer consisting of seven significant bits and a high-order sign bit. It is always byte-aligned. A scalar char variable that is not a component of a structure always occupies the high-order byte of a word of memory, and is followed by a fill byte. In general, the signed character data type does not handle eight-bit ASCII characters correctly. Use the unsigned character data type for eight-bit data; use the signed character data type for integer data with a domain of -128 to 127 (at most).

An unsigned character variable (unsigned char) is a one-byte, unsigned binary integer consisting of eight significant bits. It is never negative and always byte-aligned. A scalar unsigned char variable that is not a component of a structure always occupies the high-order byte of a word of memory, and is followed by a fill byte.

An integer variable (int) is a two-byte, signed binary integer consisting of 15 significant bits and a high-order sign bit. It is always word-aligned. This is the default data type for any variable.

An unsigned integer variable (unsigned int) is a two-byte, unsigned binary integer consisting of 16 significant bits. It is never negative and always word-aligned.

A long variable (long) is a four-byte, signed binary integer consisting of 31 significant bits and a high-order sign bit. It is always word-aligned.

An unsigned long variable (unsigned long) is a four-byte unsigned binary integer consisting of 32 significant bits. It is always positive and always word-aligned.

A floating-point variable (float) is a four-byte, word-aligned, signed real number. It can contain a value in the approximate range 8.6E-78 to 7.2E+75, with up to eight digits of precision.

A double-precision variable (double) is an eight-byte, word-aligned, signed real number. It can contain a value in the approximate range 8.6E-78 to 7.2E+75, with up to 17 digits of precision.

CONVERSIONS [6]

The following variations on baseline C operand conversion exist in MOD 400 C.

Characters and Integers [6.1]

The C compiler performs sign extension on characters. Character variables range in value from -128 to 127.

Float and Double [6.2]

The C compiler converts a double-precision variable to a floating-point variable by truncation.

Floating and Integral [6.3]

In the conversion from floating point to integral, the C compiler truncates the fraction part.

EXPRESSIONS [7]

The following variations on baseline C expressions exist in MOD 400 C.

The C compiler computes subexpressions in the order the compiler determines to be most efficient, even if the subexpressions involve side effects. The order in which side effects take place is unspecified.

Additive Operators [7.4]

If the offset to be added to (or subtracted from) a pointer is greater than 32767, an invalid pointer results, unless the offset is type long.

Shift Operators [7.5]

When a right shift is performed on a signed quantity, the sign is propagated. For instance, in the expression E1>>E2, where E1 is a signed quantity, the vacated bit positions are filled by a copy of the sign bit.

When a right shift is performed on an unsigned quantity, vacated bit positions are filled with zeros.

Assignment Operators [7.14]

There are two types of pointers: pointers to byte-aligned data (char and unsigned char), and pointers to word-aligned data (all others). A word pointer is a 32-bit DPS 6 pointer. A byte pointer is a composite, consisting of a word pointer and an int byte offset. Pointers of either type are always word aligned. Therefore, a pointer to a pointer of any type is always a word pointer. The size of a word pointer is four bytes; the size of a byte pointer is six bytes.

The C compiler also allows you to assign a pointer to an integer and an integer to a pointer; however, the conversion is reversible only if the integer is type long.

When an offset is added to (or subtracted from) a character pointer, only the integer byte offset is affected. The byte offset may eventually overflow unless the character pointer is normalized (by adjusting the word pointer and the byte offset so that the byte offset is 0 or 1). You can do this by explicitly casting the character pointer as a character pointer:

```
cp = (char *)cp;
```

This happens implicitly if the character pointer is converted to any other type.

You can use simple assignment (lvalue=expression) to copy one occurrence of a structure or union to another. The expression must have the same structure or union type as the lvalue.

DECLARATIONS [8]

The following variations on baseline C declarations exist in MOD 400 C.

Storage Class Specifiers [8.1]

The C compiler does not use register declarations for aggregate types or functions.

The C compiler accepts the first two register variables of type int, unsigned, char, or unsigned char, plus the first two register variables of type pointer. The remainder is treated as storage class auto.

You can declare data (of any type) to be storage class const. This instructs the C compiler to allocate space in the code segment rather than in the data segment. You must declare const data with initial values; once they are declared, you cannot change them.

A const identifier declared within a function has block scope. It is known and can be referenced only within the block in which it is declared. Its storage class specifier must be const or static const and it must have an initializer.

A const identifier declared outside any function has file scope. It is known and can be referenced from the point of declaration to the end of the file. If its storage class specifier is const, it can be referenced by a separately compiled function and must have an initializer. If its storage class specifier is extern const, it is a declaration that references a definition in a separately compiled function and must not have an initializer. If its storage class specifier is static const, it is a definition that can be referenced only within the current source unit and must have an initializer.

In reentrant code, an array of pointers to functions can be initialized only if it is storage class const. It can be referenced externally via the mechanism above. This is a specific instance of the reentrant code rule that pointers in data cannot be initialized to point to code (const or function) and vice versa.

Type Specifiers [8.2]

You can explicitly declare functions not returning a usable value as returning type void. For example:

```
void f1();                          /* declaration */
void f2()                           /* definition */
{
    f1()
}
```

The compiler diagnoses any expression that requires the value of a function returning void as erroneous, provided the definition or declaration of the function is in scope. If no declaration or definition is in scope, the compiler follows the rules for implicit declaration [13] and assumes type function returning int.

The enum is analogous to the scalar types of Pascal. The format is

```
enum-specifier
```

with syntax

```
enum-specifier:
   enum { enum-list }
   enum identifier { enum-list }
   enum identifier

enum-list:
   enumerator
   enum-list , enumerator

enumerator:
   identifier
   identifier = constant-expression
```

The identifier in the enum-specifier is analogous to the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { red, white, blue, green };
    .
    .
    .
enum color *cp, col;
```

makes color the enumeration tag of a type describing various colors, and then declares cp as a pointer to an object of that type, and col as an object of that type.

The identifiers in the enum-list are declared as constants, and may appear wherever constants are required. If no enumerators with = appear, then the values of the constants begin at 0 and increase by 1 as the declarations are read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

Enumeration tags and constants must all be distinct, and unlike structure tags and members, are drawn from the same set as ordinary identifiers.

Objects of a given enumeration type are regarded as having a type distinct from objects of all other types. All enumeration variables are treated as if they were int.

Structure and Union Declarations [8.5]

The C compiler only recognizes integer and character bit fields. The compiler does not initialize structures containing bit fields. The compiler assigns bit fields left to right within the word.

STATEMENTS [9]

The following variations on baseline C statements exist in MOD 400 C.

Escape Statement [9.14]

You can instruct the C compiler to pass information unchanged to the Assembly language intermediate code. The format is

```
escape "char literal"[, "char literal"]... ;
```

where "char literal" is a character string constant delimited by quotation marks. At least one string is required. Character escapes such as \t and \n are translated into ASCII characters. If a string does not end with a newline character (indicated by \n), the compiler appends one. Commas between strings are optional, but the closing semicolon is required.

The escape statement, up to and including the semicolon, is syntactically equivalent to white space.

EXTERNAL DEFINITIONS [10]

The following variations on baseline C external definitions exist in MOD 400 C.

External Function Definitions [10.1]

The C compiler converts word pointer actual parameters to character pointers by supplying a 0 byte offset. Word pointer formal parameters consider the byte offset in the calculation of formal parameter addresses but otherwise ignore it. Character actual parameters are converted to integers. Character formal parameters are converted back to characters by shifting their value to the high-order byte of the word and setting the low-order byte to 0. The entire contents of a structure or union actual parameter is passed.

COMPILER CONTROL LINES [12]

The following variations on baseline C compiler control lines exist in MOD 400 C.

Token Replacement [12.1]

The C compiler allows omission of arguments in #define statements. For example, given the statement:

```
#define list(a,b,c) a:b:c
```

subsequent uses of the identifier yield these replacements:

```
list(x,,z)  becomes            x::z
list(x, ,z) becomes            x: :z
list( , , g)        becomes     : : g
list((w,x),y,z)     becomes    (w,x):y:z
```

Text inside a string or character constant is not subject to replacement except in the token string forming the macrocall body.

File Inclusion [12.2]

An include file can contain an #include statement; this is called a "nested include."

By convention, UNIX C include files are referred to as "header files" and given a .h suffix. The C compiler does not enforce this convention.

If the filename in an #include statement is a full or relative pathname, the C compiler includes that file (or generates a fatal diagnostic error).

If the filename is expressed as

```
#include <filename>
```

then the C compiler searches these directories:

1. Directories named in -I control arguments
2. Standard library directories.

If the filename is expressed as

```
#include "filename"
```

then the C compiler searches these directories:

1. The directory of the original source unit
2. Directories named in -I control arguments
3. Standard library directories.

The MOD 400 standard C libraries are defined as >UDD>account_ID>INCLUDE and >LDD>INCLUDE.

The MOD 400 C run-time routines accept UNIX pathnames and converts them to MOD 400 equivalents; for example:

Pathname type	UNIX	MOD 400
Simple	header.h	HEADER.H
Partial	sys/params.h	SYS>PARAMS.H
Relative	../foo.h	<FOO.H
Full	/bin/hic.h	>>BIN>HIC.H

TYPES REVISITED [14]

The following variations on baseline C types exist in MOD 400 C.

Structures and Unions [14.1]

Structures can be assigned, passed as arguments to functions, and returned by functions. The types of operands involved must be the same.

If a signal occurs during the return sequence, and the same function is called reentrantly during processing of the signal, the value returned from the first call can be corrupted. The problem can occur only with signals; ordinary recursive calls work properly. (See the description of the signal function in Section 4.)

Explicit Pointer Conversions [14.4]

A char pointer-to-long-to-char pointer or word pointer-to-long-to-word pointer conversion will produce the original pointer value. A pointer-to-int-to-pointer conversion will lose the most significant bits of the pointer value and produce an invalid pointer.

PORTABILITY CONSIDERATIONS [16]

The first character is assigned to the high-order byte; a character variable is byte aligned, but a pointer to a character is word aligned.

The C compiler automatically defines the system names "mod400" or "unix" (as appropriate), the variety name "level6," and the macrocalls "__LINE__" and "__FILE__" (using double underscores). The macrocall "__LINE__" is replaced by the current line number (within the current file). The macrocall "__FILE__" is replaced by the current file name. For example, if line 10 of the program MYPROG.C contains:

```
printf("%s:%d\n",__FILE__,__LINE__);
```

it would be changed to

```
printf("%s:%d\n","MYPROG.C",10);
```

If these macrocalls appear in an include file, they are replaced by the current line number in the include file and the name of the include file, respectively.

The C compiler does not require either the Commercial or the Scientific Processor. However, if you write a program that uses date/time functions or floating-point arithmetic, you must execute it on DPS 6 hardware that includes a Scientific Processor or the Scientific Processor simulator.

Migration From MOD 400 Release 3.1 to MOD 400 Release 4.0

For a C program compiled under MOD 400 Release 4.0, using the Release 2.0 C compiler and runtime routines, and linked using the Release 4.0 Linker, there is no limit on size. Otherwise, the program is limited to 64K bytes.

To port C programs from MOD 400 Release 3.1 to Release 4.0, you must reload and relink them using the Release 2.0 C runtime routines, but you can use either the Release 3.1 or the Release 4.0 Linker. If you relink a previously reentrant program using the Release 3.1 Linker, do not use the -R or -SHARE arguments, and ignore diagnostic messages concerning $AMASK, $CMASK, and $LASTS. If you relink a reentrant program using the Release 4.0 Linker, specify the -R argument.

Register Conventions

When a C program (including run-time routines) executes, the registers it might use are:

- Data registers (R1-R7)
- Address registers (B1-B7)
- Program counter (P)
- Stack address register (T)
- CPU mode register (M1)
- SIP operating mode register (M4)
- SIP trap enable register (M5)
- System status/security register (S)
- Indicator register (I)
- Scientific accumulators (SA1-SA3).

STACK FRAME

Figure 2-1 shows the layout of a stack frame.

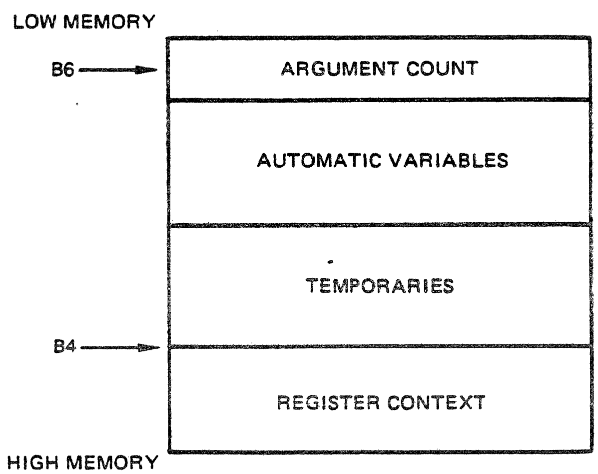

Figure 2-1. Stack Layout

Address registers B4, B6, and B7 are dedicated to unique uses. Register B6 contains a stack frame pointer to word zero of the stack frame for the currently active function. Register B4 contains a pointer to a local push-down stack within the function's stack frame. This local stack is used for compiler-generated temporary values needed during expression evaluation, and arguments to be passed to called functions. Register B7 contains a formal parameters pointer (a pointer to the argument list prepared by the calling function).

The contents of a stack frame are:

1. The number of arguments passed to the associated function, stored in the first word (at offset zero from the stack frame pointer)

2. Automatic variables, including any register variables not allocated to registers, stored beginning in the second word

3. Compiler-generated temporaries and arguments

4. A subset of the register context as it existed when the function was entered, stored at the end of the stack frame. This register context is reloaded when the the function exits.

When control transfers from one function to another and back again, register B5 contains the return address. During the transfer, register B4 contains the address of arguments being passed to the called function, and register R6 contains the argument count. The called function's entry sequence saves, in the saved register context, the contents of registers B4 and B7. Then it copies B4 into B7, and loads into B4 a pointer to the first word of the saved register context in its own stack frame.

The register context saved by a function's entry sequence is:

- Data registers R1-R5 (a few functions do not save R1)
- Indicator register (I)
- Address registers B2-B5 and B7.

Address register B6 is implicitly set and reset when a stack frame is acquired or relinquished.

The C compiler generates code assuming that the SIP operating mode register (M4) is set for double-word (32-bit) operands in SA1 and SA2 and a quad-word (64-bit) operand is set in SA3, with the corresponding memory operands set to the same precision whenever a function is entered. Register M4 is also assumed to be set to truncate mode. Functions often change M4 during their execution, but reset it to this standard state before returning or calling another function. The code generated for functions that use float or double variables assumes that any functions it calls will overwrite the contents of the scientific accumulators. (A function with no float or double variables does not use M4.)

The SIP trap enable register is not altered by the generated code; however, it is set upon entry to a function, and determines the behavior of that function in the presence of exceptions (such as exponent overflow) during floating-point operations.

REGISTER VARIABLES

Register variables are assigned as follows:

- R5 contains the first register variable of type signed int, unsigned int, signed char, or unsigned char.
- R4 contains the second register variable of type signed int, unsigned int, signed char, or unsigned char.
- B3 and R3 contain the first register variable of type pointer. R3 is used only for pointers to signed and unsigned chars.
- B2 and R2 contain the first register variable of type pointer. R3 is used only for pointers to signed and unsigned chars.

Registers R1, R6, R7, B1, B5, and any of the registers mentioned previously that are not assigned to register variables are used as working registers.

Registers R6, R7, B1, SA1, and SA3 are also used to return scalar (non-structure) values from a function to its caller. Signed and unsigned integer and signed and unsigned character values are returned in R7. Signed and unsigned long values are returned in R6 and R7. Pointers to signed and unsigned chars are returned in B1 and R7; other pointer values are returned in B1 alone. Floating-point values are returned in SA1, and double-word values are returned in SA3.

SAVED MACHINE STATE

The run-time routines save the machine state as it existed when a signal was received before calling the appropriate signal catcher. If the signal catcher returns, the saved machine state is restored to resume execution. The saved machine state consists of:

- Data registers (R1-R7)
- Address registers (B1-B7)
- Mode registers (M1-M6)
- Scientific accumulators (SA1-SA3)
- Indicator register (I)
- Scientific indicator register (SI)
- Commercial indicator register (CI).

If neither the Scientific Processor nor the Scientific Processor simulator is present, registers SA1-SA3, M4-M5, and SI are not saved and restored.

If neither the Commercial Processor nor the Commercial Processor simulator is present, register CI is not saved and restored.

Section 3
DEVELOPING C PROGRAMS

This section describes the process of program development under MOD 400. Included are descriptions of the MOD 400 C compiler, C-related commands and active functions, and C utility programs.

USING THE C COMPILER (M4_CC)

The MOD 400 C compiler can perform several operations on a C source program in sequence:

1. Preprocessing
2. Compilation
3. Assembly of compiled code
4. Prelinking
5. Linking.

You can instruct the compiler to stop after any of these operations.

The syntax of the M4_CC command is described on the following pages.

M4_CC

FORMAT:

    M4_CC file_list [ctl_arg]

ARGUMENTS:

file_list

One or more pathnames, separated by spaces, of modules the C compiler is to compile, assemble, and/or link. Files can be C source units (name.C), Assembly language source units (name.A or name.P), object units (name.O), or object directories (path). An object directory path such as <IO is equivalent to the control argument -LIB <IO. The compiler compiles, assembles, and/or links files in the order given in the command line.

You can intermix control arguments and file names in any order.

[ctl_arg]

Control arguments are processed in the order you enter them, before any files are compiled, assembled, or linked; therefore, arguments can override each other. Control arguments with parameters do not require white space separators; for example, -BU name and -BUname are both syntactically correct.

One or more of the following control arguments can be entered, in any order:

-AS

Stop after assembling the compiled code. Output is placed in the corresponding file(s) name.O.

The arguments -PP, -CO, -AS, and -LD are mutually exclusive.

-BU name

Assign name to the bound unit. Prelinker output files are named bu_name.Q and $bu_name.O; Linker output files are named bu_name.M and bu_name.

Default: The default name for C programs is A.OUT.

-CO

Stop after compiling the preprocessed code. Output is placed in the corresponding file(s) name.A.

The arguments -PP, -CO, -AS, and -LD are mutually exclusive.

-COUT out_path

Instruct the Assembler to write source listing to the file out_path.

Default: No listing is produced.

-D $\left\{\begin{array}{l}\text{name=def}\\ \text{name}\end{array}\right\}$

Enter name in the list of definitions as though either "#define name def" or "#define name 1," respectively, had occurred in the source unit. Up to 20 such names can be entered. These definitions are recorded before the first line of the C source unit is processed.

Example: M4_CC MYPROG.C -D "true=3"

-I directory

Add directory to the list of include file directories the C compiler will search. Up to eight include directories can be added. Simple filenames in #include statements are sought by the C compiler in: (1) the directory of the source unit, (2) directories named in -I control arguments, and (3) standard include directories (see Section 4).

-LD

Stop after prelinking the assembled code. The C compiler stores Linker commands in the work file bu_name.Q, and external data declarations in the Assembly language object unit $bu_name.O.

The arguments -PP, -CO, -AS, and -LD are mutually exclusive.

-LE

Produce a listing of Assembler errors and error codes only. Output is written to name.L if the -COUT argument is not specified.

Default: No listing is produced.

-LIB directory

Search LIB directory for object units during the prelinking process. If you specify this argument, the compiler will search LIB directories after the working directory, but before the directory >LDD>Z4CRT. This pathname is passed to the C prelinker. Libraries are searched in the order specified, so the placement of a -LIB control argument is significant.

Example: The control argument -LIB <IO instructs the prelinker to search the directory IO.

-LO

Produce a listing of the Assembler output. Listing is written to name.L if the -COUT argument is not specified.

-MR

Allow macrocall recursion.

-NL

Do not link the bound unit. This argument is passed to the C prelinker.

-OLDLD

Use the "old" Load utility. (See the descriptions of the Load and Old Load utilities in this section.)

-OP

Optimize the compiled object code.

-PC

Pass comments through the preprocessor.

Default: Comments are stripped out.

-PP

Stop after preprocessing the source unit(s). Preprocessing involves all source code lines beginning with #. Output is placed in the corresponding file(s) name.W.

The arguments -PP, -CO, -AS, and -LD are mutually exclusive.

-R

Generate reentrant code.

-SZ n

Request n additional 1024-word blocks of memory for linking (where n can range from 1 to 32). Use of this option can substantially reduce linking time for large programs. Refer to the MOD 400 Application Developer's Guide for suitable memory values for linking.

-U "name"

Remove name from the list of definitions, as if "#undef name" had occurred in the source unit. Up to 20 such names can be removed. These definitions are removed after the C compiler processes the -D control arguments, but before the first line of the C source unit is processed. Besides names defined by the -D control argument, only implicitly defined names like level6, unix, or mod400 can be undefined.

-UC

Change all scalars, arrays, and pointers of the type char to the type unsigned char. Treat character-string constants as if they had been declared "const unsigned char []" rather than "const char []."

EXAMPLES:

This command line compiles, assembles, and links a single C source unit, named OUT2.C. The bound unit is placed in A.OUT; the object unit is deleted by default.

```
M4_CC OUT2.C
```

This command line causes the C compiler to compile PROG1.C, ignoring the listed object files and the Linker argument:

```
M4_CC -BU PROG PROG2.O PROG3.O PROG1.C -AS
```

This command line:

```
M4_CC -BU MYPROG A.C B.A C.P C.O <OBJECT
```

causes the C compiler to:

1. Compile and assemble A.C, producing object unit A.O.
2. Assemble B.A, producing object unit B.O.
3. Assemble C.P, producing object unit C.O.
4. If there were no errors, perform a prelink edit on the object units A.O, B.O, C.O, and D.O. The compiler searches for undefined functions in the object directory <OBJECT, and then follows standard search rules.
5. Link object files A.O, B.O, C.O, D.O, and the rest, producing a bound unit named MYPROG and a link map named MYPROG.M.

This command line:

```
M4_CC <OBJECT D.O C.O B.O A.O -BU MYPROG B.A A.C C.P
```

produces the same effect as the previous example, except that the object units are produced in the order B.O, A.O, and C.O, and are linked in the order D.O, C.O, B.O, and A.O.

C-RELATED COMMANDS, ACTIVE FUNCTIONS, AND UTILITY PROGRAMS

C-related commands, active functions, and utilities are described in the following pages. They are ordered alphabetically by name. They are all available for general MOD 400 use.

NOTE

The utility program lint, commonly found on UNIX systems, is not available for MOD 400 C programs.

C Task Dump (CSICK)

Display information about a C task.

FORMAT:

```
CSICK task [ctl_arg]
```

ARGUMENTS:

task

The task to be investigated. Specify a bound unit name, a logical resource number, or a task control block address.

[ctl_arg]

One or more of the following control arguments can be entered, in any order:

{-ALL | -A}

The same as -CONTEXT -HISTORY -STACK -BOUND UNIT -HEAP -FILES -DATA -GROUP_WRITABLE_SEGMENT -GROUP_DESCRIPTOR_SEGMENT.

{-APPEND | -APP | -EXTEND}

Extend the output file. This is the default.

{-BOUND_UNIT | -BU}

List the bound units attached to the task being investigated. The bound unit name, location of the bound unit's code, and location of the bound unit's static data are given for each attached bound unit.

{-COLLECTION_WORK_AREA | -C_WORK_AREA | -CWA}

Print the C work area. The C work area is a logical extension of the task control block. It is user ring writable, and contains information required to provide the facsimile UNIX environment for C programs. The area's structure is defined in the <z4cwa.incl> header file.

{-CONSOLE_OUTPUT}
{-CO}

Direct output to the user-out file. This is the default.

When this argument is in effect, the -TRUNCATE argument is ignored..

-CONTEXT

Print the process context. The process context is:

1. The reason the process has stopped
2. The address where it stopped and its TCB address
3. The contents of its R-registers and B-registers
4. The contents of its top stack frame.

-FILES

Print the process file states. First the mapping of file descriptors onto streams is given. Then the I/O block for each stream is listed. If a given stream is open and is not attached to a MOD 400 standard file (command-in, user-in, user-out, or error-out), the file information block used to interface with the MOD 400 file system is printed following the I/O block. The buffer contents are dumped and the pathname of the file or device to which the stream is attached is also given for each open stream.

The I/O block is defined in the type definition FILE in the <z4cwa.incl> header file. The file information block is defined in the fib structure definition in the <dm_mcl.h> header file.

{-GROUP_DESCRIPTOR_SEGMENT}
{-GDS}

Print the process group descriptor segment.

{-GROUP_WRITABLE_SEGMENT}
{-GWS}

Dump the process group writable segment. The listing is preceded by a list of the work space blocks for the task group to which the process belongs. The blocks are listed in order of increasing age, including address and size.

The group writable segment is listed only for processes executing in a swappool. Work space blocks are listed regardless of memory pool type.

-HEAP

Print the heap state and the contents of the heap. The heap state is a list of the busy (allocated) blocks in the heap ordered by increasing address. For each busy block, the block's address, size (in bytes), type, and reference count are given. If a given block is not under the control of a type manager, its type is listed as "untyped" and its reference count is meaningless but usually zero.

-HISTORY

Print the process history. The process history is obtained by unwinding the process's stack in reverse order (the most recent frame first). For each frame, the return address to the caller of the activation associated with the frame, the number of and location of the arguments passed to the activation, and the location of the stack frame are printed.

$\left\{\begin{array}{l}\text{-LOGICAL\_RESOURCE\_NUMBER}\\ \text{-LRN}\end{array}\right\}$ lrn

Logical resource number, in decimal, of the task to be investigated.

$\left\{\begin{array}{l}\text{-OUTPUT\_FILE}\\ \text{-OF}\end{array}\right\}$ path

Direct the output to the file path. If path is omitted, output is placed in a working-directory file named buname.CSOUT, where buname is the bound unit name of the task's lead bound unit.

Default: Direct output to the user-out file.

-STACK

Include the contents of the stack frame itself in the process history described above. This control argument implies the -HISTORY control argument.

{-TASK_CONTROL_BLOCK} address
{-TCB}

Address of the task control block of the task to be investigated. The address must be in hexadecimal notation, in either uppercase or lowercase, optionally enclosed in parentheses, and optionally preceded by 0, x, or 0x. For example, all these addresses are valid:

054da7	X054da7
054DA7	0x054DA7
x'054Da7'	54DA7

{-TRUNCATE}
{-TC}
{-RENEW}

Truncate the output file, if it exists, before placing output in it.

This argument is ignored when the -CONSOLE_OUTPUT argument is in effect.

Default: Extend the output file.

DESCRIPTION:

The C Task Dump utility prints a detailed explanation of a task running a C program. It does this by way of formatted dumps of the process context, call history, stack frame contents, attached bound units, heap state and contents, file states, static data area, group writable segment, group descriptor segment, or any combination of these options. This utility is interactive; it accepts directives to alter the information it displays.

The utility uses the bound unit symbol table produced by the Linker when the Linker is invoked with the -SYMBOL control argument, if it can be found, to provide compile unit names and entry point names instead of numeric offset values when printing the process history. The directories listed in the PATH environment line are searched to find the bound unit symbol table file. The default PATH environment line corresponds to the directory from which the CSICK utility is loaded, followed by the directories listed in the MOD 400 loader search rules. See the C environment commands and active functions for further information on manipulating environment lines.

Interactive Mode

When the -CONSOLE_OUTPUT control argument is in effect, the utility operates interactively. The utility issues the prompt "csick:" to the user-out file at the end of each page of output and then reads a response from the user-in file. For most directives, the prompt and response cycle is then repeated.

The following directives are supported.

$\left\{\begin{array}{l}\text{ALL}\\\text{A}\end{array}\right\}$

Equivalent to the successive responses CONTEXT, STACK, BOUND_UNIT, HEAP, FILES, DATA, GROUP_WRITABLE_SEGMENT, and GROUP_DESCRIPTOR_SEGMENT.

$\left\{\begin{array}{l}\text{BOUND\_UNIT}\\\text{BU}\end{array}\right\}$

Add the list of bound units attached to the task being investigated to the list of information to be displayed.

CONTEXT

Add the process context to the list of information to be displayed.

DATA

Add the static data area to the list of information to be displayed.

$\left\{\begin{array}{l}\text{E}\\\text{..}\end{array}\right\}$ command_line

Pass command_line to the MOD 400 command processor.

FILES

Add the process's file states to the list of information to be displayed.

$\left\{\begin{array}{l}\text{GROUP\_DESCRIPTOR\_SEGMENT}\\\text{GWS}\end{array}\right\}$

Add the group descriptor segment to the list of information to be displayed.

CSICK

{GROUP_WRITABLE_SEGMENT}
{GWS}

Add the group writable segment to the list of information to be displayed.

HEAP

Add the heap state and the contents of the heap to the list of information to be displayed.

HISTORY

Add the process history to the list of information to be displayed.

{QUIT}
{Q}

Terminate the utility immediately. The prompt and response cycle is not repeated.

SKIP

Abandon the display of the set of information currently in progress. The prompt and response cycle is not repeated. To abandon the current display and request an added display, you must request the added display at the first prompt and request the skip at the second. If the current display is requested, it will be done a second time. Re-requesting the current display and then making a skip response restarts the current display.

STACK

Add the process history with the contents of the stack frames themselves included to the list of information to be displayed.

The C Task Dump utility ignores all other responses.

Using CSICK

One way to use the C Task Dump utility is to set a breakpoint at an interesting location using one of the MOD 400 debuggers. When the breakpoint is reached, execute the C Task Dump utility using the debugger's escape (E) directive.

Delete C Variable (DL_ENV)

Delete C variable.

FORMAT:

```
DL_ENV name
```

ARGUMENT:

name

The name of the variable to be deleted from the C environment. The name must begin with a dollar sign ($), underscore (_), or letter. The rest of the characters must be dollar signs, underscores, letters, or digits. The name must not be more than 32 characters long.

DESCRIPTION:

The Delete C Variable command removes a variable from the list of variables that you can pass to a C program.

ENV_DEF

Check C Variable (ENV_DEF)

Test for existence of C environment variable.

FORMAT (command):

```
ENV_DEF name
```

FORMAT (active function):

```
[ENV_DEF name]
```

ARGUMENT:

name

The name of the environment variable whose definition state is to be returned.

The name must begin with a dollar sign ($), underscore (_), or letter. The rest of the characters must be dollar signs, underscores, letters, or digits. The name must not be more than 32 characters long.

DESCRIPTION:

The Check C Variable Command tests for the existence of a C variable. This command/active function returns TRUE if name is defined as an environment variable, and FALSE if name is not defined as an environment variable.

Get C Variable (GET_ENV)

Display C variable.

FORMAT (command):

GET_ENV name

name

FORMAT (active function):

[GET_ENV name]

ARGUMENT:

name

The name of the variable to be displayed.

The name must begin with a dollar sign ($), underscore (_), or letter. The rest of the characters must be dollar signs, underscores, letters, or digits. The name must not be more than 32 characters long.

DESCRIPTION:

- The Get C Variable command/active function returns the value of the named C variable.

LIST_ENV

List C Variables (LIST_ENV)

List C variables.

FORMAT:

LIST_ENV

DESCRIPTION:

The List C Variables command lists the variables you can pass to a C program. This command writes the name and value of each C variable to the user-out file.

Load (LD)

Load program.

NOTES

1. The Load utility corresponds to the UNIX Loader, and is a link editor. It is not to be confused with the MOD 400 loader, which causes execution of bound units.

2. The version of the Load utility originally released with MOD 400 Release 3.1 is available as the bound unit OLDLD. While OLDLD is significantly slower and supports fewer features, it requires significantly less memory.

FORMAT:

```
LD buname {-LKIN} path [ctl_arg]
          {-LK  }
```

ARGUMENTS:

buname

Name of the bound unit to be created.

-LK path

Specifies the names, separated by spaces, of one or more object units to be linked into the bound unit.

Either this or the -LKIN argument is required.

This argument and the -LKIN argument are mutually exclusive.

-LKIN path

Specifies a file containing the names of one or more object units to be linked into the bound unit.

Either this or the -LK argument is required.

This argument and the -LK argument are mutually exclusive.

[ctl_arg]

One or more of the following control arguments can be entered, in any order:

-LB path

Specifies a library directory pathname. The library directory contains object units to be linked into the bound unit.

Default: The utility searches the system C library >LDD>Z4CRT.

This argument and the -LBIN argument are mutually exclusive.

-LBIN path

Specifies a file containing pathnames of library directories. Library directories contain object units to be linked into the bound unit. The utility searches directories in this order:

1. The current working directory
2. Directories specified by -LBIN argument
3. The system C library directory >LDD>Z4CRT.

This argument and the -LB argument are mutually exclusive.

-LE

Produce a listing of Assembler errors and error codes. Output is written to buname.L.

-LO

Produce a listing of Assembler output. Output is written to buname.L.

-MP path

Search load map specified by path before the default system load map. (Load maps are described under "Description.")

-NL

Do not link the bound unit. Instead, leave a Linker directive file named buname.Q.

-NO_MAIN

Generate the Linker directive LINKN Z4SUBR instead of LINKN Z4ROOT. Useful when the main program is written in FORTRAN or COBOL.

-R

Generate reentrant code (that is, separate code and data).

-SL

Suppress the Load utility banner (see the description below).

-START symbol

Generate the Linker directive START symbol. This allows multiple bound unit entry points. Also useful when the main program is written in FORTRAN or COBOL.

-SYM

Generate a symbolic history file buname.V (this argument is passed to the Linker).

-SZ n

Request n additional 1024-word blocks of memory for linking (where n may range from 1 to 44). Use of this option can substantially reduce linking time for large programs. Refer to the MOD 400 Application Developer's Guide for suitable memory-size values.

-V

Display in-progress messages as the Load utility begins each phase.

-W

Save Linker directive file.

-XREF

Create a cross-reference listing named buname.XREF.

If you specify this control argument, the Load utility calls the Sort; ignore the Sort messages.

DESCRIPTION:

The Load utility generates Linker directive files for C, FORTRAN, or COBOL source units. If you don't wish to compile and link directly from the C compiler (the default process), you can use the Load utility to:

- Link C object units
- Prelink C object units.

The Load utility generates a Linker command file with a .O suffix, initializes external storage, and performs some diagnostic checking.

When you invoke the utility, it displays this banner:

```
LOADER - n.n mm/dd/yy
```

where n.n is the release number and mm/dd/yy is the date on which the Load utility bound unit was created.

Examples:

This is the minimum valid Load command:

```
LD OUTFILE -LK NAME1
```

The following command line creates a Linker directive file for the object units NAME1 and NAME2, but stops short of creating the bound unit (by use of the -NL option). NAME1 and NAME2 are in the directories named in the file LBDIR. External names are resolved using the load map file MAPFILE.

```
LD OUTFILE -LK NAME1 NAME2 -LBIN LIBDIR -NL -MP MAPFILE
```

Load Maps:

A load map resolves external names when an object unit contains more than one function definition, or when an object unit has a name different from some function it contains. For example, the default load map (>LDD>Z4CRT>Z4LDMP) contains these entries:

```
COS     SIN
BRK     Z4BRK
```

These entries specify that the cos function is located in the object unit SIN.O, and that the brk function is located in the object unit Z4BRK.O.

When the Load utility locates an external name in an object file, it searches the load map (the one you specified in the -MP argument, then the default system load map >LDD>Z4CRT>Z4LDMP). If an entry for that external name is found, the Load utility replaces it with the corresponding name from the load map. It searches for an object unit by that name in (1) the working directory, (2) any directories specified in a -LB argument, and (3) the system directory >LDD>Z4CRT.

To create your own load map, first create the file. A load map must be a dynamic indexed sequential file with fixed-length records. Here are sample commands to create such a file and its index:

```
CR MYLOADMAP -DYN -LRSZ 256 -CISZ 512
CX MYLOADMAP.X MYLOADMAP -KLOC 1 -KSZ 6
```

MYLOADMAP is the data file; MYLOADMAP.X is the load map. To use the load map, specify -MP MYLOADMAP.X in the Load command line.

Then you can edit the data file to add entries, one per line. Entries must correspond to this C structure:

```
struct  loadmap_entry {
        char external_name[6];
        char separator[2];
        char object_file[6];
};
```

where external_name is the external name (the key field indicated in the Create Index command above), separator is two spaces, and object_file is the object unit name (NOT including the .O suffix). Therefore, the external names you enter must be from one to six characters long, left-justified, and blank-filled to six characters; two spaces must follow; and the object unit name must be from one to six characters long. All names must be in upper case.

Once you have populated the load map, you can print the data file to view the entries in their original order, or the index file to view the entries in key order.

OLDLD

Old Load (OLDLD)

"Old" Load utility.

NOTES

1. The Load utility corresponds to the UNIX Loader, and is a link editor. It is not to be confused with the MOD 400 loader, which causes execution of bound units.

2. This is the version of the Load utility originally released with MOD 400 Release 3.1. While OLDLD is is significantly slower and supports fewer features than LD, it requires significantly less memory.

FORMAT:

```
OLDLD buname {-LKIN} path [ctl_arg]
             {-LK  }
```

ARGUMENTS:

buname

Name of the bound unit to be created.

-LK path

Specifies the names, separated by spaces, of one or more object units to be linked into the bound unit.

Either this or the -LKIN argument is required.

This argument and the -LKIN argument are mutually exclusive.

-LKIN path

Specifies a file containing the names of one or more object units to be linked into the bound unit.

Either this or the -LK argument is required.

This argument and the -LK argument are mutually exclusive.

[ctl_arg]

One or more of the following control arguments can be entered, in any order:

-LB path

Specifies a library directory pathname. The library directory contains object units to be linked into the bound unit.

Default: The utility searches the system C library >LDD>Z4CRT.

This argument and the -LBIN argument are mutually exclusive.

-LBIN path

Specifies a file containing pathnames of library directories. Library directories contain object units to be linked into the bound unit. The utility searches directories in this order:

1. The current working directory
2. Directories specified by -LBIN argument
3. The system C library directory >LDD>Z4CRT.

This argument and the -LB argument are mutually exclusive.

-NL

Do not link the bound unit. Instead, leave a Linker directive file named buname.Q.

-SZ n

Request n additional 1024-word blocks of memory for linking (where n may range from 1 to 44). Use of this option can substantially reduce linking time for large programs. Refer to the MOD 400 Application Developer's Guide for suitable memory-size values.

-XREF

Create a cross-reference listing named buname.XREF.

DESCRIPTION:

The Old Load utility prepares and uses Linker directive files for C source units. You can create bound units from C source units directly from the C compiler (the default action); link C object units using the Load utility; or stop after prelinking C object units using the Load utility. The Load utility generates a Linker command file with a .Q suffix, initializes external storage, and performs some diagnostic checking.

Example:

This is the minimum valid Old Load command:

```
LD OUTFILE -LK NAME1
```

The following command line creates a Linker directive file for the object units NAME1 and NAME2, but stops short of creating the bound unit (by use of the -NL argument). NAME1 and NAME2 are in the directories named in the file LBDIR.

```
LD OUTFILE -LK NAME1 NAME2 -LBIN LIBDIR -NL
```

Set C Variable (SET_ENV)

Set C variable.

FORMAT:

```
SET_ENV name [value]
```

ARGUMENTS:

name

The name of the variable to be set. The name argument must begin with a dollar sign ($), an underscore (_), or a letter. The rest of the characters must be dollar signs, underscores, letters, or digits. The name argument must not be more than 32 characters long.

[value]

The value to which name is set. If the value argument is omitted, a null string is assumed.

DESCRIPTION:

The Set C Variable command sets a value which you can then pass to a C program.

Section 4
THE C STANDARD LIBRARY

This section lists the standard functions and subroutines provided with the MOD 400 C compiler.

The routines provided with the C compiler attempt to present C programs with the same interface they would enjoy under UNIX. However, due to the inherent differences in the two operating systems, some routines are altered, have restrictions not found on UNIX, or are not supported at all. For instance, routines that involve pathnames adhere to MOD 400 pathname conventions, not UNIX pathname conventions; the process-management functions (for example, fork) are available only to tasks running in a MOD 400 swappool; and functions involving the UNIX "super user" (for example, getpass and some elements of kill) are not allowed under MOD 400 at all. Also excluded are these functions:

- Data base
- Multiplexed-file
- Multiprecision integer arithmetic
- Plotter I/O
- Packet driver
- Interprocess communication
- Semaphore
- Archive
- X.25.

Table 4-1 lists C system functions and subroutines, sorted by name; Table 4-2 lists the same functions sorted by function group. Table 4-3 lists commonly used UNIX system functions (taken from System V UNIX) not supported under MOD 400 C.

The MOD 400 standard C include directories are located in >UDD>account_ID>INCLUDE and >LDD>INCLUDE.

Table 4-1. MOD 400 C Standard Library (Sorted by Name)

Name	Function	Function Group
a641	Convert base-64 ASCII to long	String
abort	Generate IOT fault	Process
abs	Absolute value of integer	Mathematical
access	Determine accessibility of file	File control
acos	Arc cosine	Mathematical
alarm	Schedule signal after interval	Process
alloc	Main memory allocation	Storage
asctime	Convert time to ASCII	System
asin	Arc sin	Mathematical
atan	Arc tangent	Mathematical
atan2	Arc tangent	Mathematical
atof	Convert ASCII to floating-point	String
atoi	Convert ASCII to integer	String
atol	Convert ASCII to long integer	String
brk	Change memory allocation	Storage
bsearch	Binary search	String
calloc	Main memory allocation	Storage
ceil	Ceiling function	Mathematical
chdir	Change working directory	File control
chown	Change owner	Process
clearerr	File status inquiry	Input/output
close	Close file	File control
cos	Cosine	Mathematical
cosh	Hyperbolic cosine	Mathematical
creat	Create new file	File control
crypt	DES encryption	System
ctime	Convert date/time to ASCII	System
dup	Duplicate open file descriptor	File control

Table 4-1 (cont). MOD 400 C Standard Library (Sorted by Name)

Name	Function	Function Group
ecvt	Output conversion	String
encrypt	DES encryption	System
endgrent	Close group file	File control
endpwent	Close password file	System
equal_name	Equal-names convention	File control
erf	Return error function of arg	Mathematical
erfc	Return 1-erf(x)	Mathematical
errno	Error message number	System
execl	Execute a file	Process
execle	Execute a file	Process
execlp	Execute a file	Process
execv	Execute a file	Process
execve	Execute a file	Process
execvp	Execute a file	Process
exit	Terminate a process	Process
exp	Exponential function	Mathematical
fabs	Absolute value of real value	Mathematical
fclose	Close a file	Input/output
fcntl	Control over open files	File control
fcvt	Output conversion	String
fdopen	Open a file	Input/output
feof	File status inquiry	Input/output
ferror	File status inquiry	Input/output
fflush	Flush a file	Input/output
fgetc	Get character from word or file	Input/output
fgets	Get string from file	Input/output
fileno	File status inquiry	Input/output
find_file	Find a file	File control
floor	Floor function	Mathematical
fmod	Return remainder function (a/b)	Mathematical
fopen	Open a file	Input/output
fork	Spawn a new process	Process
fprintf	Formatted output conversion	Input/output
fputc	Put character or word on file	Input/output
fputs	Put string on file	Input/output
fread	Buffered binary input	Input/output
free	Main memory allocation	Storage
freopen	Reopen a file	Input/output
frexp	Split into mantissa and exponent	Mathematical
fscanf	Formatted input conversion	Input/output
fstat	Get file status	File control
fwrite	Buffered binary output	Input/output

Table 4-1 (cont). MOD 400 C Standard Library (Sorted by Name)

Name	Function	Function Group
gamma	Log absolute value gamma function	Mathematical
gcvt	Output conversion	String
getc	Get character from word or file	Input/output
getchar	Get character from word or file	Input/output
getcwd	Get current working directory	File control
getdir	Get pathname of system directory	System
getegid	Get effective group ID	Process
getenv	Get environment name	Process
geteuid	Get effective user ID	Process
getgid	Get group ID	Process
getgrent	Get group file entry	File control
getgrgid	Get group file entry	File control
getgrnam	Get group file entry	File control
getlogin	Get login name	Process
getopt	Get option letter from arg	String
getpgrp	Get process group	Process
getpid	Get process ID	Process
getppid	Get parent process ID	Process
getpwent	Get password record entry	System
getpwnam	Get password record by login name	System
getpwuid	Get password record by user ID	System
getptcb	Get parent TCB	System
gets	Get string from file	Input/output
getr	Get record	Input/output
gettcb	Get TCB	System
getuid	Get user ID	Process
getw	Get word from file	Input/output
gmtime	Convert to Greenwich Mean Time	System
hypot	Euclidean distance	Mathematical
init_mem	Initialize memory	Storage
isalnum	Character classification	String
isalpha	Character classification	String
isascii	Character classification	String
isascii8	Character classification	String
isatty	Get name of terminal	System
iscntrl	Character classification	String
isdigit	Character classification	String
isgraph	Character classification	String
islower	Character classification	String
isprint	Character classification	String
ispunct	Character classification	String
isspace	Character classification	String
isupper	Character classification	String
isxdigit	Character classification	String

Table 4-1 (cont). MOD 400 C Standard Library (Sorted by Name)

Name	Function	Function Group
j0	Bessel function	Mathematical
j1	Bessel function	Mathematical
jn	Bessel function	Mathematical
kill	Send signal to process	Process
l3tol	Convert 3-byte integer to long	Storage
l64a	Convert long to base-64 ASCII string	String
ldexp	Split into mantissa and exponent	Mathematical
lgdiv	Long divide	Mathematical
lgmul	Long multiply	Mathematical
lgrem	Long remainder	Mathematical
link	Link to a file	File control
localtime	Convert date/time to local time	System
log	Natural logarithm	Mathematical
log10	Common logarithm	Mathematical
longjmp	Non-local goto	System
lsearch	linear search	File control
ltol3	Convert long integer to 3-byte	Storage
malloc	Main memory allocator	Storage
mcl	Execute MOD 400 macrocall	System
memccpy	Memory-to-memory copy	Storage
memchr	Point to character in memory	Storage
memcmp	Compare memory areas	Storage
memcpy	Memory-to-memory copy	Storage
memset	Initialize memory	Storage
mktemp	Make unique file name	File control
modf	Split into mantissa and exponent	Mathematical
open	Open file	File control
pause	Stop until signal	Process
perror	Print system error message	System
pipe	Interprocess communication	Process
posr	Position file record pointer	Input/output
pow	Power function	Mathematical
printf	Formatted output conversion	Input/output
pthto6	Convert UNIX pathname to MOD 400	System
putc	Put character or word on file	Input/output
putchar	Put character or word on file	Input/output
putr	Put record on a file	Input/output
puts	Put string on file	Input/output
putw	Put word on file	Input/output
qsort	Quicker sort	System

Table 4-1 (cont). MOD 400 C Standard Library (Sorted by Name)

Name	Function	Function Group
rand	Random number generator	Mathematical
read	Read from a file	Input/output
realloc	Reallocate memory	Storage
runl	Create new process	Process
runlp	Create new process	Process
runv	Create new process	Process
runvp	Create new process	Process
same_file	Compare pathnames	File control
sbrk	Change memory allocation	Storage
scanf	Formatted input conversion	Input/output
send_sig	Send signal to process	Process
setbuf	Assign buffering to a file	Input/output
setgrent	Rewind group file	Process
setjmp	Prepare for non-local goto	System
setkey	DES encryption	System
setprint	Set print attribute of stream	Process
setpwent	Rewind password file	System
signal	Catch signal	Process
sin	Sine	Mathematical
sinh	Hyperbolic sine	Mathematical
sleep	Suspend execution for interval	Process
smopen	Open for block read/write	File control
smread	Read block	File control
smwrit	Write block	File control
sprintf	Formatted output conversion	Input/output
sqrt	Square root	Mathematical
srand	Random number generator	Mathematical
sscanf	Formatted input conversion	Input/output
star_check	Validate star name	File control
star_match	Validate and match star name	File control
star_name	List star name matches	File control
stat	Get file status	File control
strcat	Character-string concatenation	String
strchr	First C occurrence	String
strcmp	Compare	String
strcpy	Copy	String
strcspn	Compare length of strings	String
strlen	Length	String
strncat	Concatenate N characters	String
strncmp	Compare N characters	String
strncpy	Copy N characters	String
strpbrk	Find first S_1 in S_2	String
strrchr	First C occurrence	String
strspn	Length of S_1 substr of S_2 chars	String
strtok	Token separator	String
swab	Swap bytes	String

Table 4-1 (cont). MOD 400 C Standard Library (Sorted by Name)

Name	Function	Function Group
sys_errlist	Vector of system error messages	System
sys_nerr	Largest system error message number	System
system	Execute a command line	System
tan	Tangent	Mathematical
tanh	Hyperbolic tangent	Mathematical
time	Get time	Process
tmpnam	Create temporary file name	File control
toascii	Character translation	String
tolower	Character translation	String
toupper	Character translation	String
ttyname	Get name of terminal	System
tzset	Set time zone	System
ucf_defc	Create file	File control
ucf_defr	Create file	File control
ucf_finish	Create file	File control
ucf_init	Create file	File control
uldiv	Long unsigned divide	Mathematical
ulrem	Long unsigned remainder	Mathematical
umemchr	Point to character in memory	Storage
umemcmp	Compare memory areas	Storage
umemcpy	Memory-to-memory copy	Storage
umemset	Initialize memory	Storage
ungetc	Push character back into input file	Input/output
unlink	Remove directory entry	File control
wait	Wait for process to terminate	Process
write	Write on file	Input/output
y0	Bessel function	Mathematical
y1	Bessel function	Mathematical
yn	Bessel function	Mathematical

Table 4-2. MOD 400 C Routines (Sorted by Function Group)

Group	Name	Function
File control	access	Determine accessibility of file
	chdir	Change working directory
	close	Close file
	creat	Create new file
	dup	Duplicate open file descriptor
	endgrent	Close group file
	equal_name	Equal-names convention
	fcntl	Control over open files
	find_file	Find a file
	fstat	Get file status
	getcwd	Get current working directory
	getgrent	Get group file entry
	getgrgid	Get group ID
	getgrnam	Get group name
	link	Link to a file
	lsearch	linear search
	mktemp	Make unique file name
	open	Open file
	same_file	Compare pathnames
	stat	Get file status
	tmpnam	Create name for temporary file
	ucf_defc	Create file
	ucf_defr	Create file
	ucf_finish	Create file
	ucf_init	Create file
	unlink	Remove directory entry
Input/output	clearerr	File status inquiry
	fclose	Close a file
	fdopen	Open a file
	feof	File status inquiry
	ferror	File status inquiry
	fflush	Flush a file
	fgetc	Get character from word or file
	fgets	Get string from file
	fileno	File status inquiry
	fopen	Open a file
	fprintf	Formatted output conversion
	fputc	Put character or word on file
	fputs	Put string on file
	fread	Buffered binary input
	freopen	Reopen a file
	fscanf	Formatted input conversion
	fwrite	Buffered binary output

Table 4-2 (cont). MOD 400 C Routines (Sorted by Function Group)

Group	Name	Function
Input/output (cont)	getc	Get character from word or file
	getchar	Get character from word or file
	getr	Get record from file
	gets	Get string from file
	getw	Get word from file
	posr	Position file record pointer
	printf	Formatted output conversion
	putc	Put character or word on file
	putchar	Put character or word on file
	putr	Put record on file
	puts	Put string on file
	putw	Put word on file
	read	Read from file
	scanf	Formatted input conversion
	setbuf	Assign buffering to file
	sprintf	Formatted output conversion
	sscanf	Formatted input conversion
	ungetc	Push character back into input file
	write	Write on file
Mathematical	abs	Absolute value of integer
	acos	Arc cosine
	asin	Arc sin
	atan	Arc tangent
	atan2	Arc tangent
	ceil	Ceiling function
	cos	Cosine
	cosh	Hyperbolic cosine
	erf	Return error function of arg
	erfc	Return 1-erf(x)
	exp	Exponential function
	fabs	Absolute value of real value
	floor	Floor function
	fmod	Return remainder function (a/b)
	frexp	Split into mantissa and exponent
	gamma	Log absolute value gamma function
	hypot	Euclidean distance
	j0	Bessel function
	j1	Bessel function
	jn	Bessel function
	ldexp	Split into mantissa and exponent
	lgdiv	Long divide
	lgrem	Long remainder
	lgmul	Long multiply
	log	Natural logarithm
	log10	Common logarithm
	modf	Split into mantissa and exponent

Table 4-2 (cont). MOD 400 C Routines (Sorted by Function Group)

Group	Name	Function
Mathematical (cont)	pow	Power function
	rand	Random number generator
	sin	Sine
	sinh	Hyperbolic sine
	sqrt	Square root
	srand	Random number generator
	tan	Tangent
	tanh	Hyperbolic tangent
	uldiv	Long unsigned divide
	ulrem	Long unsigned remainder
	y0	Bessel function
	y1	Bessel function
	yn	Bessel function
Process	abort	Generate IOT fault
	alarm	Schedule signal after interval
	chown	(No function)
	execl	Execute a file
	execle	Execute a file
	execlp	Execute a file
	execv	Execute a file
	execve	Execute a file
	execvp	Execute a file
	exit	Terminate a process
	fork	Spawn a new process
	getegid	Get effective group ID
	getenv	Get environment name
	geteuid	Get effective user ID
	getgid	Get group ID
	getlogin	Get login name
	getpgrp	Get process group
	getpid	Get process ID
	getppid	Get parent process ID
	getuid	Get user ID
	kill	Send signal to process
	pause	Stop until signal
	pipe	Interprocess communication
	runl	Create new process
	runlp	Create new process
	runv	Create new process
	runvp	Create new process
	send_sig	Send signal to process
	setgrent	Rewind group file
	setprint	Set print attribute of stream
	signal	Catch signal
	sleep	Suspend execution for interval
	time	Get time
	wait	Wait for process to terminate

Table 4-2 (cont). MOD 400 C Routines (Sorted by Function Group)

Group	Name	Function
Storage	alloc	Main memory allocation
	brk	Change memory allocation
	calloc	Main memory allocation
	free	Main memory allocation
	init_mem	Initialize memory
	l3tol	Convert 3-byte to long integer
	ltol3	Convert long to 3-byte integer
	malloc	Main memory allocator
	memccpy	Memory-to-memory copy
	memchr	Point to character in memory
	memcmp	Compare memory areas
	memcpy	Memory-to-memory copy
	memset	Initialize memory
	realloc	Reallocate memory
	sbrk	Change memory allocation
	umemchr	Point to character in memory
	umemcmp	Compare memory areas
	umemcpy	Memory-to-memory copy
	umemset	Initialize memory
String	a64l	Convert base-64 ASCII to long
	atof	Convert ASCII to floating point
	atoi	Convert ASCII to integer
	atol	Convert ASCII to long integer
	bsearch	Binary search
	ecvt	Output conversion
	fcvt	Output conversion
	gcvt	Output conversion
	getopt	Get option letter from arg
	isalnum	Character classification
	isalpha	Character classification
	isascii	Character classification
	isascii8	Character classification
	iscntrl	Character classification
	isdigit	Character classification
	isgraph	Character classification
	islower	Character classification
	isprint	Character classification
	ispunct	Character classification
	isspace	Character classification
	isupper	Character classification
	isxdigit	Character classification
	l64a	Convert long to base-64 ASCII string
	strcat	Character-string concatenation
	strchr	First C occurrence
	strcmp	Compare
	strcpy	Copy
	strcspn	Compare length of strings

Table 4-2 (cont). MOD 400 C Routines (Sorted by Function Group)

Group	Name	Function
String (cont)	strlen	Length
	strncat	Concatenate N characters
	strncmp	Compare N characters
	strncpy	Copy N characters
	strpbrk	Find first S_1 in S_2
	strrchr	First C occurrence
	strspn	Length of S_1 substring of S_2
	strtok	Token separator
	swab	Swap bytes
	toascii	Character conversion
	toascii8	Character conversion
	tolower	Character conversion
	_tolower	Character conversion
	toupper	Character conversion
	_toupper	Character conversion
System	asctime	Convert time to ASCII
	crypt	DES encryption
	ctime	Convert date/time to ASCII
	encrypt	DES encryption
	endpwent	Close password file
	errno	Error message number
	getdir	Get pathname of system directory
	getptcb	Get parent TCB
	getpwent	Get password record entry
	getpwnam	Get password record by login name
	getpwuid	Get password record by user ID
	gettcb	Get TCB
	gmtime	Convert date/time to Greenwich Time
	isatty	Get name of terminal
	localtime	Convert date/time to local time
	longjmp	Non-local goto
	mcl	Execute MOD 400 macrocall
	perror	Print system error message
	pthto6	Convert UNIX pathname to MOD 400
	qsort	Quicker sort
	setjmp	Prepare for non-local goto
	setkey	DES encryption
	setpwent	Rewind password file
	sys_errlist	Vector of system error messages
	sys_nerr	Largest system error message number
	system	Execute a command line
	ttyname	Get name of terminal
	tzset	Set time zone

Table 4-3. C Routines Not Supported

Name	Function
acct	Enable or disable process accounting
assert	Program verification
chmod	Change mode of file
chroot	Change root directory
clock	Report CPU time used
ctermid	Get terminal ID
cuserid	Get user ID
dbminit	Data base subroutine
delete	Data base subroutine
dial	Dial external line
edata	End of program initialized data location
end	End of program data location
etext	End of program code location
fetch	Data base subroutine
firstkey	Data base subroutine
fseek	Reposition a file
ftell	Reposition a file
ftw	File tree walk
getpass	Read password
gsignal	Get signal
hcreate	Create heap
hdestroy	Destroy heap
hsearch	Search heap
logname	Login name of user
lseek	Change file currency
matherr	Math routine error handler
mknod	Make node (directory or file)
monitor	Prepare execution profile
mount	Mount volume
mpx et al	Create and manipulate multiplexed files
nextkey	Data base subroutine
nice	Change priority of a process
nlist	Get entries from name list

*

Table 4-3 (cont). C Routines Not Supported

Name	Function
pclose	Close a pipe
plock	Process lock
popen	Open a pipe to/from process
profil	Execution profile
ptrace	Trace a process
putpwentry	Write password entry
regcmp	Compile regular expression
regx	Execute regular expression
setgid	Set group ID
setpgrp	Set process group
setuid	Set user ID
sgetl	Get long numeric
sig	Signal
sputl	Put long numeric
stime	Set time
store	Data base subroutine
strtol	Convert string to long integer
stty	Set terminal characteristics
synch	Update superblock
tdelete	Delete tree
tell	Change file currency
times	Get process times
tmpfile	Create temporary file
tmpnam	Temporary name
tsearch	Search tree
twalk	Walk tree
tzname	Get time zone
* unmount	Dismount volume
utime	Set file time stamps

C SUPPORT OF MOD 400 FILE TYPES

C supports sequential files with most functions. The getr and putr functions support relative, random, dynamic, and indexed files.

The creat function creates a sequential file. Sequential processing of pre-existing string-relative files will be compatible with UNIX. The functions smopen, smread, and smwrit support block-mode processing. The lseek function is not supported.

SUBROUTINES AND LIBRARIES

C subroutines and libraries include input/output and mathematical functions. While these functions are not directly callable from C, you can use these functions with include statements of the form:

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

Functions in the math library may return conventional values 0 or HUGE (largest size precision floating number) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable errno is set to the value EDOM or ERANGE.

The descriptions of some functions refer to the null pointer (NULL). This value will not match that of any legitimate pointer, so many functions that return pointers return it, for example, to indicate an error. NULL is defined in <stdio.h> as (int*)0; you can include your own definition if you are not using <stdio.h>.

The standard I/O package consists of the stdio.h header file and a set of functions. The inline macrocalls getc and putc handle characters quickly. The macrocalls getchar, putchar, and the higher level routines fgetc, fgets, fprintf, fputc, fputs, fread, fscanf, fwrite, gets, getw, printf, puts, putw, and scanf all use getc and putc; they can be freely intermixed.

A file with associated buffering is declared to be a pointer to a defined type FILE. The fopen function creates certain descriptive data for a file and returns a pointer to designate the file in all further transactions. Normally, there are three open files with constant pointers declared in the "include" file and associated with the standard open files:

```
stdin  -- Standard input file (MOD 400 user-in)
stdout -- Standard output file (MOD 400 user-out)
stderr -- Standard error file (MOD 400 error-out).
```

An integer constant EOF (-1) is returned when a function encounters the end of a file or an error (see the individual descriptions for details).

Any application that uses this package must include the header file of pertinent macrocall definitions, as follows:

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

The functions and constants mentioned in the input/output functions are declared in that "include" file and need no further declaration. The constants and the following "functions" are macrocalls (redeclaration of these names is perilous):

- clearerr
- feof
- fileno
- getc
- getchar
- putc
- putchar.

TRAPS AND SIGNALS

Generally, MOD 400 traps are mapped to their UNIX equivalents, to provide an emulation of the UNIX environment. After catching a signal in UNIX, a program can continue as if the signal had not been sent merely by returning from the signal catcher (as opposed to calling exit.) In MOD 400, this is not always possible. For this reason, these (MOD 400) traps cause the task to be terminated upon return from the signal catcher:

Trap	Meaning
0	Cleanup
3	Uninstalled SIP operation
12	Recursive remote descriptors
13	Unprivileged use of privileged operation
15	Reference to unavailable resource
17	Bus parity or memory error
24	Uncorrectable memory error or Megabus error
32	CIP/SIP reference to protected memory
33	Invalid SIP argument

MOD 400 traps will be mapped into UNIX signals as described in Table 4-4.

Table 4-4. MOD 400 Trap Support of UNIX Signals

MOD 400 Trap	UNIX Signal
0 Cleanup	15 Software termination signal
1 MCL instruction	Filtered out by support routines
1 PI command	2 Interrupt
2 BRK instruction	5 Trace trap
2 Trace trap	5 Trace trap
3 Uninstalled SIP instruc.	4 Invalid instruction
4 RSU	Trap disabled
5 Other uninstalled instr.	4 Invalid instruction
6 Integer arith. overflow	Trap disabled
7 SIP divide by zero	8 Floating-point exception
8 SIP exponent overflow	8 Floating-point exception
9 Stack underflow	Used by support routines only
10 Stack overflow	Used by support routines only
12 Recursive remote descriptor usage	4 Invalid instruction
13 Unprivileged use of privileged instruction	4 Invalid instruction
14 Out of ring bracket	11 Segmentation violation
15 Unavailable resource	11 Segmentation violation
16 Program error	4 Invalid instruction
17 Bus/memory parity error	10 Bus error
19 SIP exponent underflow	8 Floating-point exception
20 SIP program error	4 Invalid instruction
21 SIP significance error	8 Floating-point exception
22 SIP precision error	8 Floating-point exception
23 SIP/CIP unavail. resource	11 Segmentation violation
24 Uncorrectable memory error	10 Bus error
25 CIP divide by 0	Not applicable
26 CIP bad specification	Not applicable
27 CIP bad character	Not applicable
28 CIP truncation error	Not applicable
29 CIP arithmetic overflow	Not applicable
30 CIP self-test fault	Trap disabled
31 SIP self-test fault	Trap disabled
32 CIP/SIP reference to protected memory	11 Segmentation violation
33 Invalid SIP argument	4 Invalid instruction
48 BREAK key	Trap disabled
49 Unwind command	3 Quit
51 Intergroup signal	Depends on message content
53 Power-fail restart	19 Power-fail restart
Not applicable	1 Hangup
Not applicable	6 IOT instruction[a]
Not applicable	7 EMT instruction[a]

[a]On some processors

Table 4-5 lists software-generated signals. Note that "pid" is the process ID.

Table 4-5. Software-Generated Signals

C Calling Sequence	Meaning
kill (pid, 0)	Test signal, always ignored
kill (pid, SIGHUP)	1 Hangup
kill (pid, SIGINT)	2 Interrupt
kill (pid, SIGQUIT)	3 Quit
kill (pid, SIGILL)	4 Invalid instruction
kill (pid, SIGTRAP)	5 Trace trap
kill (pid, SIGIOT)	6 IOT instruction[a]
kill (pid, SIGEMT)	7 EMT instruction[a]
kill (pid, SIGFPE)	8 Floating-point exception
kill (pid, SIGKILL)	9 Kill
kill (pid, SIGBUS)	10 Megabus error
kill (pid, SIGSEGV)	11 Segmentation violation
kill (pid, SIGSYS)	12 Bad argument to function
kill (pid, SIGPIPE)	13 Write to pipe having no readers
kill (pid, SIGALRM)	14 Alarm clock
alarm (delta)	14 Alarm clock (after delta secs)
kill (pid, SIGTERM)	15 Terminate
kill (pid, SIGUSR1)	16 User defined signal 1
kill (pid, SIGUSR2)	17 User defined signal 2
kill (pid, SIGCLD)	18 Death of a child
kill (pid, SIGPWR)	19 Power-fail restart

[a]On some processors

In UNIX, the interrupt and quit signals are sent to every process in the process group that is not ignoring the signal. Processes created to run a command in the background (asynchronously) are created with these signals being ignored. Processes created to run a command in the foreground (synchronously) are created with default handling of these signals unless otherwise specified via the trap command. All other processes inherit the handling of these (and all other) signals from their parent.

To emulate this handling of interrupt and quit signals, the converted program interrupt trap is broadcast to the entire process group instead of being sent only to the receiving process. MOD 400 broadcasts the unwind trap to the entire process group.

ERROR RETURNS

Most functions have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always -1; the individual descriptions specify the details.

Reporting Errors Via errno

A UNIX error number is returned in the external integer variable errno. The variable errno is not cleared on successful calls, so it should be tested only after an error has been indicated.

Reporting Errors Via m4_errno

If an error is returned by a MOD 400 system service macrocall, the MOD 400 error number is returned via the external integer variable m4_errno. Any time an error is reported (for example, by returning a null pointer), both of these variables are set. The errno external variable is set to the appropriate UNIX error number and m4_errno is set to the MOD 400 error code that led the runtime routine to report the error. In situations when there is no MOD 400-detected error leading to the error being reported, m4_errno is set to 1800 hexadecimal plus the UNIX error number.

UNIX Errors

All of the possible error numbers are not listed in each function description because many errors are possible for most of the calls. The following is a complete list of the error numbers, manifest constants, and names as defined in <error.h>.

1 EPERM Not owner.

In UNIX, this error typically indicates an attempt to modify a file in some way forbidden except to its owner or super-user.

2 ENOENT No such file or directory.

This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a pathname does not exist.

3 ESRCH No such process.

No process can be found corresponding to that specified by the process ID in kill.

4 EINTR Interrupted process.

An asynchronous signal (such as interrupt or quit), which the user has elected to catch, has occurred during a function. If execution is resumed after processing the signal, it appears as if the interrupted function returned this error condition. This is a UNIX error only; it never occurs in MOD 400.

5 EIO I/O error.

Some physical I/O error. This error may in some cases occur on a call following the one to which it actually applies.

6 ENOIO No such device or address.

In UNIX, this occurs when I/O on a special file refers to a device that does not exist, or is beyond the limits of the device. It may also occur when, for example, a tape drive is not online or no disk pack is loaded on a drive.

7 E2BIG Argument list too long.

An argument list longer than 5120 characters is presented to a member of the exec family.

8 ENOEXEC Exec format error.

In UNIX, this occurs when a request is made to execute a file which, although it has the appropriate access, does not start with a valid magic number. This is a UNIX error only; it never occurs in MOD 400.

9 EBADF Bad file number.

A file descriptor refers to no open file, a read request is made to a file that is open only for writing, or a write request is made to a file that is only open for reading.

10 ECHILD No children.

A wait was executed by a process that has no existing child processes; or by a process already waiting for all its children.

11 EAGAIN No more processes.

A fork failed because you are not allowed to create any more processes.

12 ENOMEM Not enough memory.

During an exec, brk, sbrk, or other function, a program asks for more space than MOD 400 can supply. This is not a temporary condition; the maximum space is a system parameter. The error can also occur if the arrangement of text, data, and stack segments requires too many segments.

13 EACCES Permission denied.

An attempt has been made to access a file to which you have insufficient access.

14 EFAULT Bad address.

MOD 400 has encountered a hardware fault while using a function argument.

15 ENOTBLK Block device required.

In UNIX, this occurs when a nonblock file is mentioned where a block device is required; for example, in mount.

16 EBUSY Mount device is busy.

In UNIX, this occurs when an attempt is made to mount a device that is already mounted, or an attempt is made to dismount a device on which there is an active file (open file or current directory). It also occurs if an attempt is made to enable accounting when it is already enabled.

17 EEXIST File already exists.

An existing file is mentioned in an inappropriate context; for example, link.

18 EXDEV Cross-device link.

In UNIX, this occurs when a link to a file on another device is attempted.

19 ENODEV No such device.

An attempt has been made to apply an inappropriate function to a device; for example, read a write-only device.

20 ENOTDIR Not a directory.

A file is specified where a directory is required, for example in a path prefix or as an argument to chdir.

21 EISDIR Is a directory.

An attempt has been made to write on a directory.

22 EINVAL Invalid argument.

Some invalid argument has occurred; for example, mentioning an undefined signal in signal, or kill. This error is also set by the mathematical functions.

23 ENVILE File table overflow.

The system's table of open files is full, and temporarily no more opens can be accepted.

24 EMFILE Too many open files.

No process can have more than 20 file descriptors open at a time.

25 ENOTTY Not a typewriter.

The device is not a terminal.

26 ETXTBSY Text file busy.

In UNIX, this occurs when an attempt has been made to execute a pure procedure program that is currently open for writing (or reading); or an attempt has been made to open for writing a pure procedure program that is being executed.

27 EFBIG File too large.

In UNIX, this occurs when the size of a file exceeds the maximum file size or ULIMIT. In MOD 400, this occurs when the file size limit is exceeded.

28 ENOSPC No space left on device.

During a write to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek.

In UNIX, this occurs when an lseek has been issued to a pipe.

30 EROFS Read-only file system.

An attempt was made to modify a file or directory on a device mounted read-only; that is, with the write-protect switch set.

31 EMLINK Too many links.

In UNIX, this occurs on an attempt to make more than the maximum number of links (1000) to a file.

32 EPIPE Broken pipe.

In UNIX, this occurs when a write has been attempted on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 EDOM Math argument not in function's domain.

The argument of a function in the math package is out of the domain of the function.

34 ERANGE Math function's result too large.

The value of a function in the math package is not representable within machine precision.

35 ENOMSG No message of desired type.

An attempt was made to receive a message of a type that does not exist on the specified queue. This is a UNIX error only; it never occurs in MOD 400.

36 EIDRM Identifier removed.

This error is returned to processes that resume execution due to the removal of an identifier from the file system's name space. This is a UNIX error only; it never occurs in MOD 400.

MOD 400 Extensions

Error numbers greater than 36 are MOD 400 extensions to the basic UNIX set.

59 ENOSWP Fork attempted from a non swap memory pool.

The fork function was attempted by a process executing in a non-swappool.

60 EBADEQ Bad syntax in an equal name.

An equal name that does not conform to the equal-name syntax was presented to the equal-name function.

61 EUNMEQ Unmatched percent sign or equal sign in an equal name.

An equal name presented to the equal-name function contains a percent or equal sign that has no matching component in the source name.

62 EBIGEQ Equal name too long.

An equal name and source name presented to the equal-name function resulted in a target name more than 12 characters long.

64 EBADSTAR Bad syntax in star name.

A star name that does not conform to the star-name syntax was presented to one of the star-name functions.

ABANDONING A PROCESS

The procedure for abandoning a process is only available to assembly language routines. It is invoked by calling the c_rtn macrocall with the parameter dead. This macrocall generates code to call the dead routine which in turn abandons the process by sending a trap 16 (10 hexadecimal) to itself. The B3, B5 and B7 registers in the error message displayed by the MOD 400 default trap handler have the following values:

B3 = The return address to dead's caller; somewhere in Z4DMA when heap management abandons the process.

B5 = The return address to the caller of dead's caller.

B7 = The address of the argument list passed to dead's caller. This is not meaningful if dead's caller was called without arguments.

Registers B1, B6, R3, R4 and R5 contain whatever dead's caller left in them. Normally, B6 points to the stack frame of dead's caller. This is the top stack frame since dead does not use a stack frame itself.

Registers B2, B4, R1, R2, R6 and R7 are destroyed while abandoning the process.

The heap management routines print one of three error messages on the MOD 400 error-out file before abandoning the process. These messages are:

1834 Heap Format error. Process abandoned.

This message indicates that a block listed in the heap's free list is not marked "free" in its status word. For this message, B1 points at the status word of the block involved and R3 contains its quad-word offset within the heap.

1835 Block format error detected by free. Process abandoned.

This message indicates that free, grow, or realloc was given a pointer that: (1) doesn't point somewhere within the heap, (2) isn't congruent to 2 modulo N (counting in bytes), or (3) is marked free in its status word (grow and realloc only). Under MOD 400 Release 3.1, N is 8; under MOD 400 Release 4.0, N is 16. See the character pointer pointed to by B7 for the cause.

1836 Attempt to free an already free block. Process abandoned.

This indicates that free was called with a pointer to a block already marked "free" in its status word. No check of the heap's free list has been made. B1 contains the address of the block's status word. Notice that free checks for block format error before checking for block already free.

The stack overflow trap handler abandons the process if it is unable to obtain sufficient memory to eliminate the condition. If the process is running in a swappool, the memory is obtained by expanding the stack segment. In a non-swappool, the memory is obtained from the heap (this may cause a heap overflow). Before attempting to obtain the needed memory, a check is made to see if the stack size limitation placed on the bound unit when it was linked would be exceeded. If it would, the process is abandoned without attempting to obtain any additional memory. (The stack size limitation is 256 times $ISS plus $INCMX times two to the $INCSS power words where $ISS, $INCMX and $INCSS are Linker value definitions created by the LD command.)

Before abandoning the process, the stack overflow trap handler prints the following message on the MOD 400 error-out file:

1833 Stack overflow. Unable to grow stack further.

The trap 16 error message displayed by the MOD 400 default trap handler has the following values:

B3 = The return address to dead's caller; somewhere in Z4TRAP when the process is abandoned due to stack overflow.

B5 = The address of the instruction following the ACQ instruction which caused the stack overflow trap.

B7 = The address of the A word in the process' copy of the trap save area.

The initialize memory routine init_mem sends the SIGSYS signal to the calling process when it is given (char *) 0 as the pointer to the memory to be initialized and the size given is not zero. If the SIGSYS signal is ignored or its signal catcher returns, the initialize memory routine calls the dead routine to abandon the process. The values displayed for B5 and B7 in the error message from the MOD 400 default trap handler identify the initialize memory routine's caller and the arguments passed by the caller.

The long jump routine sends the SIGSYS signal to the calling process when it is given an environment pointer which does not in fact point to a saved environment. If the SIGSYS signal is ignored or its signal catcher returns, the long jump routine calls the dead routine to abandon the process. The values displayed for B5 and B7 in the error message from the MOD 400 default trap handler identify the long jump routine's caller and the arguments passed by the caller.

RUN-TIME ROUTINES

The rest of this section describes the run-time routines (either functions or macrocalls) available under MOD 400 C. The descriptions are arranged alphabetically by routine name. Refer to Tables 4-1 and 4-2 for a complete list of routines.

a64l

Convert between long and base-64 ASCII.

FORMAT:

```
long a64l (s)
char *s;
```

ARGUMENTS:

s

Value to be converted.

DESCRIPTION:

The a64l function is used to maintain numbers stored in base-64 ASCII. This is a notation by which long integers can be represented by up to six characters; each character represents a digit in a radix-64 notation.

The characters used to represent "digits" are . for 0, / for 1, 0 through 9 for 2-11, A through Z for 12-37, and a through z for 38-63.

RETURN VALUE:

This function returns a long value.

RELATED FUNCTIONS:

l64a.

abort

Terminate a C program.

FORMAT:

```
int abort ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The abort function causes an IOT signal to be sent to its own process. The default signal catcher causes program termination with a memory dump.

It is possible for abort to return control if SIGIOT is caught or ignored. In this case, the value returned is that of the kill function.

DIAGNOSTICS:

The following message is displayed:

> Unclaimed signal 6 (SIGIOT) from process pid received by process pid.

where pid is the process ID.

RELATED FUNCTIONS:

exit, signal.

abs

Integer absolute value.

FORMAT:

```
int abs (i)
int i;
```

ARGUMENTS:

i

Integer value whose absolute value is to be returned.

DESCRIPTION:

The abs function returns the absolute value of its integer operand.

RELATED FUNCTIONS:

fabs.

access

Determine access rights or existence of a file.

FORMAT:

```
int access (path, amode)
char *path;
int amode;
```

ARGUMENTS:

path

Pointer to a pathname naming a file.

amode

Bit pattern constructed as a sum of the following:

04 -- Read
02 -- Write
01 -- Execute (search)

DESCRIPTION:

The access function checks the access rights of the named file according to the bit pattern contained in the amode argument.

The file has access checked with respect to the read, write, and execute mode bits.

No access to the file is indicated if the information request of the file system returns an error. Error codes returned are associated with the MOD 400 Get File Access Rights system service macrocall.

RETURN VALUE:

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of -1 is returned. The variable errno is set to indicate the UNIX error.

acos

Arc cosine function.

FORMAT:

```
# include <math.h>

double acos (x)
double x;
```

ARGUMENTS:

x

Double value of the cosine.

DESCRIPTION:

The acos function returns the arc cosine in the range 0 to pi.

DIAGNOSTICS:

Arguments of magnitude greater than 1 cause acos to return value 0.

RELATED FUNCTIONS:

asin, atan, atan2, cos, sin, tan.

alarm

Set a process alarm clock.

FORMAT:

```
unsigned alarm (sec)
unsigned sec;
```

ARGUMENTS:

sec

Number of seconds until alarm.

DESCRIPTION:

The alarm function instructs the calling process's alarm clock to send the signal SIGALRM to the calling process after the number of real-time seconds specified by the sec argument have elapsed; see signal.

Alarm requests are not stacked; successive calls replace the calling task's alarm clock.

If sec is 0, any previously made alarm request is canceled.

RETURN VALUE:

The alarm function returns the amount of time, possibly 0, previously remaining in the calling process's alarm clock.

DIAGNOSTICS:

If alarm is unable to set the alarm clock for any reason, errno and m4_errno are set to indicate the reason and the hexadecimal value FFFF is returned. The first call to alarm may fail because the task is unable to obtain memory for the alarm clock or because it is unable to create an auxiliary task to listen for the alarm to go off. Reasons for failure are:

- Lack of group work segment memory--errno set to ENOMEM
- Lack of available LRN--errno set to EAGAIN

RELATED FUNCTIONS:

pause, signal.

alloc

The alloc function is a synonym for malloc. See the description of the malloc function.

asctime

Convert date and time to ASCII.

FORMAT:

```
# include <time.h>

    char *asctime (tm)
    struct tm *tm;
    extern long timezone;
    extern int daylight;
    extern char *tzname[2];
```

ARGUMENTS:

tm

Time, in military notation.

DESCRIPTION:

The asctime function converts the components of the time to ASCII and returns a pointer to a 26-character string in the following form (all fields have constant width):

```
Fri Aug 10 10:24:54 1984\n\0
```

The structure declaration from the include file is:

```
struct tm   {
       int      tm_sec;
       int      tm_min;
       int      tm_hour;
       int      tm_mday;
       int      tm_mon;
       int      tm_year;
       int      tm_wday;
       int      tm_yday;
       int      tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday - 0), year - 1900, day of year (0-365), and a flag that is nonzero if daylight saving time is in effect.

The external long variable timezone contains the difference, in seconds, between GMT and local standard time (in EST, timezone is 5*60*60); the external variable daylight is nonzero if and only if the standard U.S. daylight savings time conversion should be applied.

If the environment variable TZ is not present, the asctime function assumes the local time zone is the same as the system time zone. The external variable daylight is set to zero in this case.

If TZ is present, the asctime function uses it to determine the local time zone. The value of TZ must be a time zone acronym, a time offset, and an optional daylight-savings time zone acronym.

- The time zone acronym is up to four characters long.
- The time offset represents the difference between local time in the designated time zone and GMT. The difference is represented by a string of digits with an optional leading minus sign (for locations east of Greenwich, England) and with an optional trailing .5 (for locations some odd number of half-hours from Greenwich).
- The optional daylight savings time zone acronym is up to four characters long.

For example, the setting for Boston would be EST5EDT.

Setting TZ changes the values of the external variables timezone and daylight; in addition, time zone acronyms contained in the external variable tzname are set:

```
char *tzname[2] = {"EST ", "EDT "};
```

NOTE

The return values point to static data whose contents are overwritten by each call.

RELATED FUNCTIONS:

ctime, gmtime, localtime, time, tzset; see also the list_stz and set_stz commands.

asin

Arc sine function.

FORMAT:

```
# include <math.h>

double asin (x)
double x;
```

ARGUMENTS:

x

Double-precision value of the sin.

DESCRIPTION:

The asin function returns the arc sine in the range -pi/2 to pi/2.

DIAGNOSTICS:

Arguments of magnitude greater than 1 cause asin to return value 0.

RELATED FUNCTIONS:

acos, atan, atan2, cos, sin, tan.

atan

Arc tangent function.

FORMAT:

```
# include <math.h>

double atan (x)
double x;
```

ARGUMENTS:

x

Double-precision value of the tangent.

DESCRIPTION:

The atan function returns the arc tangent of x in the range -pi/2 to pi/2.

RELATED FUNCTIONS:

acos, asin, atan2, cos, sin, tan.

atan2

Arc tangent of y/x.

FORMAT:

```
# include <math.h>

double atan2 (y, x)
double x, y;
```

ARGUMENTS:

x

Double-precision value.

y

Double-precision value.

DESCRIPTION:

The atan2 function returns the arc tangent of y/x in the range -pi to pi.

RELATED FUNCTIONS:

acos, asin, atan, cos, sin, tan.

atof

Convert ASCII to floating point.

FORMAT:

```
double atof (aptr)
char *aptr;
```

ARGUMENTS:

aptr

A string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional e or E following by an optionally signed integer.

DESCRIPTION:

The atof function converts a string to floating-point representation. The first unrecognized character ends the string.

NOTE

There are no provisions for overflow.

RELATED FUNCTIONS:

atoi, atol, scanf.

atoi

Convert ASCII to integer.

FORMAT:

```
int atoi (aptr)
char *aptr;
```

ARGUMENTS:

aptr

A string of tabs and spaces, then an optional sign, then a string of digits.

DESCRIPTION:

The atoi function converts a string to integer representation. The first unrecognized character ends the string.

NOTE

There are no provisions for overflow.

RELATED FUNCTIONS:

atof, atol, scanf.

atol

Convert ASCII to long.

FORMAT:

```
long atof (aptr)
char *aptr;
```

ARGUMENTS:

aptr

A string of tabs and spaces, then an optional sign, then a string of digits.

DESCRIPTION:

The atol function converts a string to long integer representation. The first unrecognized character ends the string.

NOTE

There are no provisions for overflow.

RELATED FUNCTIONS:

atof, atoi, scanf.

brk

Change break segment space allocation.

FORMAT:

```
int brk (endds)
char *endds;
```

ARGUMENTS:

endds

New address of break value.

DESCRIPTION:

The brk function dynamically changes the amount of space allocated for the calling process's break segment. The change is made by resetting the process's break value. The break value is the address of the first location beyond the end of the break segment. The amount of allocated space increases as the break value increases.

The brk function sets the break value to endds and changes the allocated space accordingly.

DIAGNOSTICS:

The brk function fails without making any change in the allocated space if such a change would result in more space being allocated than can be mapped to the segments available to the calling process [ENOMEM].

RETURN VALUE:

Upon successful completion, brk returns a value of 0. Otherwise, errno and m4_errno are set to indicate the error and -1 is returned.

RELATED FUNCTIONS:

exec family, sbrk

NOTES

1. The first call to brk creates a break segment. It may be a giant segment (larger than 128K characters). If this segment cannot be created for any reason, errno is set to ENOMEM and -1 is returned. The segment number in its formal parameter determines where in the caller's address space the break segment resides.

2. When a C task runs outside of a swappool, MOD 400 allocates memory from the task's memory pool instead of creating a segment; subsequent calls cannot increase the size of the break segment.

bsearch

bsearch

Binary search.

FORMAT:

```
char *bsearch (key, base, nelem, width, compar)
char *key;
char *base;
int nelem, width;
int (*compar)();
```

ARGUMENTS:

key

Pointer to the datum to be located in the table.

base

Pointer to the base of the table.

nelem

Number of elements in the table.

width

Width of an element in characters.

compar

Name of the comparison routine.

DESCRIPTION:

The bsearch function is a binary search routine generalized from Knuth Algorithm B. The table must be previously sorted in increasing order.

The comparison routine indicated by compar is called with two character-pointer (char *) arguments that point to the table elements being compared. This comparison routine must return an integer less than, equal to, or greater than 0 depending on whether the first argument is less than, equal to, or greater than the second.

RETURN VALUE:

The bsearch routine returns a pointer into the table indicating the location of the table element that matches the key.

DIAGNOSTICS:

A null pointer is returned if the key cannot be found in the table.

RELATED FUNCTIONS:

lsearch, qsort.

calloc

Heap memory allocation.

FORMAT:

```
char *calloc (nelem, elsize)
unsigned nelem, elsize;
```

ARGUMENTS:

nelem

Number of elements.

elsize

Size of each element in characters.

DESCRIPTION:

The calloc function allocates space for an array of elements. The space is initialized to zeros.

RETURN VALUE:

The calloc function returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS:

If the heap contains insufficient memory to allocate the requested block and cannot be expand sufficiently, the variable errno is set to ENOMEM, the variable m4_errno is set to 1800 (hexadecimal) plus ENOMEM, and (char *) 0, a null character pointer, is returned.

RELATED FUNCTIONS:

free, malloc, realloc.

ceil

Ceiling function.

FORMAT:

```
double ceil (x)
double x;
```

ARGUMENTS:

x

Double-precision value to be compared.

DESCRIPTION:

The ceil function returns the smallest integer not less than x.

RELATED FUNCTIONS:

abs, fabs, floor, fmod.

chdir

Change working directory.

FORMAT:

```
int chdir (path)
char *path;
```

ARGUMENTS:

path

Pointer to the pathname of a directory.

DESCRIPTION:

The chdir function changes the current working directory to the named directory.

RETURN VALUE:

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the variable errno is set to indicate the error.

chown

Change owner.

FORMAT:

```
int chown (path, owner, group)
char *path;
int owner, group;
```

DESCRIPTION:

The chown function has no effect in MOD 400. It is provided to be able to satisfy calls to the function and not require their deletion.

RETURN VALUE:

The success value of 0 is always returned.

clearerr

File status inquiry -- clear error indicator.

FORMAT:

```
# include <stdio.h>

clearerr (file)
FILE *file;
```

ARGUMENTS:

file

File pathname.

DESCRIPTION:

The clearerr function resets the error indication on the named file.

The clearerr function is a macrocall; it cannot be redeclared.

RELATED FUNCTIONS:

feof, ferror, fileno, fopen, open.

close

Close a file.

FORMAT:

```
# include <stdio.h>

int close (fildes)
int fildes;
```

ARGUMENTS:

fildes

File descriptor obtained from a create, dup, fcntl, or pipe function.

DESCRIPTION:

The close function closes and deletes a file. The close function closes the file descriptor indicated by fildes. A shared file is not removed until the last user executes a close.

RETURN VALUE:

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned. The variable errno is set to indicate the error.

DIAGNOSTICS:

The close function fails if fildes is not a valid, open file descriptor.

RELATED FUNCTIONS:

creat, dup, exec family, fcntl, open, pipe.

cos

Cosine function.

FORMAT:

```
# include <math.h>

double cos (x)
double x;
```

ARGUMENTS:

x

Double-precision value of the angle in radians.

DESCRIPTION:

The cos function returns the cosine of a radian argument. The magnitude of the argument should be checked by the caller to ensure that the result is meaningful.

RELATED FUNCTIONS:

acos, asin, atan, atan2, sin, tan.

cosh

Hyperbolic function.

FORMAT:

```
# include <math.h>

double cosh (x)
double x;
```

ARGUMENTS:

x

Double-precision value.

DESCRIPTION:

The cosh function computes the hyperbolic cosine function for real arguments.

DIAGNOSTICS:

The cosh function returns a huge value of appropriate sign when the correct value would overflow.

RELATED FUNCTIONS:

sinh, tanh.

creat

Create a new file or rewrite an existing one.

FORMAT:

```
int creat (path, mode)
char *path;
int mode;
```

ARGUMENTS:

path

File pathname.

mode

File access--ignored (see below).

DESCRIPTION:

The creat function creates a new sequential file or prepares to rewrite an existing file named by the pathname pointed to by the path argument.

The mode argument (which in UNIX sets file access) is ignored. Access control list (ACL) rights for the file are determined by whatever ACLs and Common Access Control Lists (CACLs) currently apply to the file.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged.

RETURN VALUE:

Upon successful completion, the file descriptor (a non-negative integer) is returned and the file is opened for writing. The file descriptor is set to remain open across exec functions (see fcntl). The file pointer is set to the beginning of the file. No process can have more than 20 files open simultaneously.

Otherwise, a value of -1 is returned, and the variable errno is set to indicate the error.

RELATED FUNCTIONS:

close, dup, open, read, write.

crypt

DES encryption.

FORMAT:

```
char *crypt (key, salt)
char *key, *salt;
```

ARGUMENTS:

key

User's typed password.

salt

Two-character string chosen from the lowercase letters, the uppercase letters, the digits 0 through 9, the slash (/), and the period (.).

DESCRIPTION:

The crypt function is a password encryption routine based on the National Bureau of Standards Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

The salt string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string.

RETURN VALUE:

The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

RELATED FUNCTIONS:

crypt, encrypt, setkey.

NOTE

The return value points to static data that is overwritten by each call.

ctime

Convert date and time to ASCII.

FORMAT:

```
# include <time.h>

char *ctime (clock)
long *clock;
```

ARGUMENTS:

clock

Long integer pointer to the time in seconds since midnight GMT, Jan. 1, 1970 (such as returned by time).

DESCRIPTION:

The ctime function converts a time into ASCII and returns a pointer to a 26-character string in the following form (all fields have constant width):

```
Sat Aug 10 10:24:54 1985\n\0
```

The structure declaration from the include file is:

```
struct tm   {
      int     tm_sec;
      int     tm_min;
      int     tm_hour;
      int     tm_mday;
      int     tm_mon;
      int     tm_year;
      int     tm_wday;
      int     tm_yday;
      int     tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday - 0), year - 1900, day of year (0-365), and a flag that is nonzero if daylight saving time is in effect.

The external long variable timezone contains the difference, in seconds, between GMT and local standard time (in EST, timezone is 5-60-60); the external variable daylight is nonzero if, and only if, the standard U.S. daylight savings time conversion should be applied.

NOTE

The return values point to static data whose contents are overwritten by each call.

RELATED FUNCTIONS:

asctime, gmtime, localtime, time, tzset; see also the list_stz and set_stz commands.

dup

Duplicate an open file descriptor.

FORMAT:

```
int dup (fildes)
int fildes;
```

ARGUMENTS:

fildes

File descriptor obtained from a creat, open, dup, fcntl, or pipe function.

DESCRIPTION:

The dup function duplicates an open file descriptor.

RETURN VALUE:

This function returns a new file descriptor having the following in common with the original:

- Same open file
- Same file pointer (that is, both file descriptors share one file pointer)
- Same access (read, write, execute).

The new file descriptor is set to remain open across exec functions (see fcntl).

The file descriptor returned is the lowest one available.

If an error occurs, a value of -1 is returned.

RELATED FUNCTIONS:

creat, close, exec, fcntl, open, pipe.

ecvt

Output conversion.

FORMAT:

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;
```

ARGUMENTS:

value

Double-precision value to be converted.

ndigit

Number of digits in output string.

decpt

Pointer to position of the decimal point relative to the beginning of the string (negative means to the left of the returned digits).

sign

If the sign of the result is negative, the word pointed to by sign is nonzero; otherwise it is zero.

DESCRIPTION:

The ecvt function converts a value to a null-terminated string of ndigit digits and returns a pointer thereto. If the sign of the result is negative, the word pointed to by sign is nonzero; otherwise it is zero. The low-order digit is rounded.

NOTE

The return values point to static data whose contents are overwritten by each call.

RELATED FUNCTIONS:

fcvt, gcvt, printf.

encrypt

DES encryption.

FORMAT:

```
encrypt (block, edflag)
char *block;
int edflag;
```

ARGUMENTS:

block

Sixty-four-character binary array.

edflag

If the value of edflag is 0, the argument is encrypted; if nonzero, it is decrypted.

DESCRIPTION:

The encrypt function is based on the National Bureau of Standards Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search. The encrypt function provides access to the actual DES algorithm.

The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by the setkey function.

RELATED FUNCTIONS:

crypt, setkey.

NOTE

The return value points to static data that is overwritten by each call.

endgrent

End group record entry.

FORMAT:

```
# include <grp.h>

void endgrent ()
```

ARGUMENTS:

None.

DESCRIPTION:

A call to endgrent has the effect of making the next call to getgrent a "first" call.

RELATED FUNCTIONS:

getgrent, getgrgid, getgrnam, getlogin, getpwent, group, setgrent.

endpwent

Close password file.

FORMAT:

```
# include <pwd.h>

void endpwent ()
```

ARGUMENTS:

None.

DESCRIPTION:

A call to the endpwent function has the effect of making the next call to getpwent a "first" call.

RELATED FUNCTIONS:

getpwent, getpwnam, getpwuid, setpwent.

errno

System error message number.

FORMAT:

```
extern int errno;
```

ARGUMENTS:

None.

DESCRIPTION:

The external variable errno is set when errors occur but not cleared when nonerroneous calls are made. The variable errno can be used as an index into the table of system error messages (see sys_errlist) to get the message string without the newline character.

RELATED FUNCTIONS:

perror, sys_errlist, sys_nerr.

equal_name

equal_name

Equal-names convention.

FORMAT:

```
int equal_name ();

    equal_name (equal, file, target)
    equal_name (equal, file)
    equal_name (equal)

    unsigned char file [13], equal [13], target [13];
```

ARGUMENTS:

equal

Input equal name.

file

Input file name.

target

Output target name which is constructed.

DESCRIPTION:

This subroutine implements the MOD 400 equal-names convention. A target name is constructed by combining components and subcomponents from a file name and an equal name which are supplied as arguments.

The target name is constructed whenever the file name is present. The absence of a target argument merely means it is not returned to the caller.

An equal name is constructed according to the following rules:

1. An equal name is a file name. Therefore it is composed of a string of 12 or fewer ASCII printing graphics, none of which can be the greater than (>), less than (<), circumflex (^), exclamation point (!) or slash mark (/) characters.

2. An equal name is composed of one or more nonnull components. This means an equal name cannot begin or end with a period (.) and cannot contain two or more consecutive periods.

3. Each percent sign (%) appearing in an equal name component is treated as a special character.

4. Each equal sign (=) appearing in an equal name component is treated as a special character.

5. An equal name component consisting of only a double (==) or triple equal sign (===) is treated as a special component.

An equal name maps characters from a given file name into another file name, the target name, according to the following rules:

1. Each percent sign (%) in the equal name represents the single character in the corresponding component and character position of the given file name. An error occurs if the corresponding character does not exist.

2. An equal sign (=) in an equal name component represents the corresponding component of the given file name. An error occurs if the corresponding component does not exist. An error also occurs if an equal sign appears in a component that also contains a percent sign. Only one equal sign can appear in each equal name component, except for the double or triple equal sign component, as noted below.

3. The double equal sign (==) component of an equal name represents all components of the given file name that have no other corresponding components in the equal name. Often, the double equal sign component represents more than one component of the given file name. If so, the number of components represented by the entire equal name is the same as the number of components in the file name. When the the equal name contains the same number of components or more components than the file name, the double equal sign is meaningless and, therefore, ignored. Only one double equal sign component can appear in an equal name.

4. The triple equal sign (===) component of an equal name represents the entire given file name. The triple equal sign component is used to add

components to a name. Only one triple equal sign component may appear in an equal name and no other component of that equal name may contain percent signs or equal signs.

RETURN VALUE:

The return value is one of the following:

2 The equal name is ===, ==, ==.=, or =.== and the target name, if the file name is present, has been constructed without error.

1 The equal name is some other valid equal name containing at least one = or % and the target name, if the file name is present, has been constructed without error.

0 The equal name is valid but contains no equal or percent signs, and the target name, if the file name is present, has been constructed without error.

-1 An error has been detected.

DIAGNOSTICS:

If an error is detected, the external variable errno is set to one of the values listed below. The external variable m4_errno is set to that value plus 1800 (hexadecimal).

60 The equal name has an invalid format.

61 There was no letter or component in the file name that corresponds to a % or = in the equal name. A null string is used for the missing letter or component in the target name that is returned. This can only occur when the file name is present.

62 The target name to be constructed is longer than 12 characters. Only the first 12 characters are returned. This can only occur when the file name is present.

erf

Error function.

FORMAT:

```
# include <math.h>

double erf (x)
double x;
```

ARGUMENT:

x

Double-precision value.

DESCRIPTION:

The erf function returns the error function of x. The error function is:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)\, dt$$

RELATED FUNCTIONS:

erfc, exp.

erfc

Complimentary error function.

FORMAT:

```
# include <math.h>

double erfc (x)
double x;
```

ARGUMENT:

x

Double-precision value.

DESCRIPTION:

The erfc function is defined as:

$$\mathrm{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)\, dt$$

This function is provided because of the extreme loss of relative accuracy of the error function erf(x) for large values of x.

RELATED FUNCTIONS:

erfc, exp.

execl

Execute a bound unit.

FORMAT:

```
int execl(path,arg0,arg1,...,argn,(unsigned char *) 0)
unsigned char *path, *arg0, *arg1, ..., *argn;
```

ARGUMENTS:

path

Pointer to a pathname that identifies the new process bound unit.

arg_0, arg_1, ..., arg_n

Pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, at least arg_0 must be present and point to a string that is the same as path (or its file-name component).

DESCRIPTION:

The execl function transforms the calling process into a new process. The new process is constructed from an ordinary bound unit called the new process bound unit. There can be no return from a successful exec because the calling process is overlaid by the new process.

When a C program is executed, it is called as follows:

```
int main (argc, argv, envp)
int argc;
unsigned char **argv, **envp;
```

where argc is the argument count and argv is an array of character pointers to the arguments themselves. By convention, argc is at least one and argv[0] points to a string containing the name of the file.

This function allows MOD 400 pathname syntax to be used in the path formal parameter in addition to the UNIX syntax. In fact, MOD 400 and UNIX syntax can be mixed in a given path; for example, <LIST/PROG.L is permitted.

A pointer to the environment of the calling process is placed in the global cell:

```
extern unsigned char **environ;
```

It is used to pass the environment of the calling process to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set. For those file descriptors that remain open, the file currency (read or write) is unchanged.

Signals set to be ignored by the calling process, or set to terminate the calling process, remain so set. Signals set to be caught by the calling process are set to terminate the new process.

The new process also inherits the following attributes from the calling process:

- Process ID
- Parent process ID
- Process group ID
- TTY group ID
- Time left until an alarm signal
- Current working directory
- Root directory
- File mode creation mask
- File size limit
- Task self-delete switch.

The execl function fails and returns to the calling process if:

- One or more components of the pathname do not exist [ENOENT].
- A directory-name component of path is not a directory [ENOTDIR].
- List access is denied for a directory named in path [EACCES].
- The new process bound unit is not a bound unit, or the calling process lacks execute access to it [EACCES].
- The new process requires more memory than is allowed [ENOMEM].

- The number of characters in the argument list for the new process is greater than 5120 characters [E2BIG].
- The path, argv, or envp argument points to an invalid address [EFAULT].

RETURN VALUE:

If execl returns to the calling process, an error has occurred; the return value is -1, and the variables m4_errno and errno are set to indicate the error.

RELATED FUNCTIONS:

execle, execv, execve, exit, fork, getenv; see also the dl_env, get_env, list_env, and set_env commands.

execle

Execute a bound unit.

FORMAT:

```
int execle(path,arg0,arg1,...,argn,(unsigned char *)0),envp)
unsigned char *path, *arg0, *arg1, ..., *argn, *envp [];
```

ARGUMENTS:

path

Pointer to a pathname that identifies the new process bound unit.

arg_0, arg_1, ..., arg_n

Pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, at least arg_0 must be present and point to a string that is the same as path (or its file name component).

envp

Array of character pointers to null-terminated strings. These strings constitute the environment for the new process. The array is terminated by a null character pointer.

DESCRIPTION:

The execle function transforms the calling process into a new process. The new process is constructed from an ordinary bound unit called the new process bound unit. There can be no return from a successful exec because the calling process is overlaid by the new process.

When a C program is executed, it is called as follows:

```
int main (argc, argv, envp)
int argc;
unsigned char **argv, **envp;
```

where argc is the argument count and argv is an array of character pointers to the arguments themselves. By convention, argc is at least one and argv[0] points to a string containing the name of the file.

This function allows MOD 400 pathname syntax to be used in the path formal parameter in addition to the UNIX syntax. In fact, MOD 400 and UNIX syntax can be mixed in a given path; for example, <LIST/PROG.L is permitted.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set. For those file descriptors that remain open, the file currency (read or write) is unchanged.

Signals set to be ignored by the calling process, or set to terminate the calling process, remain so set. Signals set to be caught by the calling process are set to terminate the new process.

The new process also inherits the following attributes from the calling process:

- Process ID
- Parent process ID
- Process group ID
- TTY group ID
- Time left until an alarm signal
- Current working directory
- Root directory
- File mode creation mask
- File size limit
- Task self-delete switch.

The execle function fails and returns to the calling process if:

- One or more components of the pathname do not exist [ENOENT].
- A directory-name component of path is not a directory [ENOTDIR].
- List access is denied for a directory named in path [EACCES].
- The new process bound unit is not a bound unit, or the calling process lacks execute access to it [EACCES].
- The new process requires more memory than is allowed [ENOMEM].
- The number of characters in the argument list for the new process is greater than 5120 characters [E2BIG].
- The path, argv, or envp argument points to an invalid address [EFAULT].

RETURN VALUE:

If execle returns to the calling process, an error has occurred; the return value is -1, and the variables m4_errno and errno are set to indicate the error.

RELATED FUNCTIONS:

execl, execv, execve, exit, fork, getenv; see also the dl_env, get_env, list_env, and set_env commands.

execv

Execute a bound unit.

FORMAT:

```
int execv (path, argv)
unsigned char *path, *argv [];
```

ARGUMENTS:

path

Pointer to a pathname that identifies the new process bound unit.

argv

Array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, argv must have at least one member, and it must point to a string that is the same as path (or its file name component). The array is terminated by a null character pointer.

DESCRIPTION:

The execv function transforms the calling process into a new process. The new process is constructed from an ordinary bound unit called the new process bound unit. There can be no return from a successful exec because the calling process is overlaid by the new process.

When a C program is executed, it is called as follows:

```
int main (argc, argv, envp)
int argc;
unsigned char **argv, **envp;
```

where argc is the argument count and argv is an array of character pointers to the arguments themselves. By convention, argc is at least one and argv[0] points to a string containing the name of the file.

This function allows MOD 400 pathname syntax to be used in the path formal parameter in addition to the UNIX syntax. In fact, MOD 400 and UNIX syntax can be mixed in a given path; for example, <LIST/PROG.L is permitted.

A pointer to the environment of the calling process is placed in the global cell:

```
extern unsigned char **environ;
```

It is used to pass the environment of the calling process to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set. For those file descriptors that remain open, the file currency (read or write) is unchanged.

Signals set to be ignored by the calling process, or set to terminate the calling process, remain so set. Signals set to be caught by the calling process are set to terminate the new process.

The new process also inherits the following attributes from the calling process:

- Process ID
- Parent process ID
- Process group ID
- TTY group ID
- Time left until an alarm signal
- Current working directory
- Root directory
- File mode creation mask
- File size limit
- Task self-delete switch.

The execv function fails and returns to the calling process if:

- One or more components of the pathname do not exist [ENOENT].

- A directory-name component of path is not a directory [ENOTDIR].

- List access is denied for a directory named in path [EACCES].

- The new process bound unit is not a bound unit, or the calling process lacks execute access to it [EACCES].

- The new process requires more memory than is allowed [ENOMEM].

- The number of characters in the argument list for the new process is greater than 5120 characters [E2BIG].
- The path, argv, or envp argument points to an invalid address [EFAULT].

RETURN VALUE:

If execv returns to the calling process, an error has occurred; the return value is -1, and the variables m4_errno and errno are set to indicate the error.

RELATED FUNCTIONS:

execl, execle, execve, exit, fork, getenv; see also the dl_env, get_env, list_env, and set_env commands.

execve

Execute a bound unit.

FORMAT:

```
int execve (path, argv, envp);
unsigned char *path, *argv [], *envp [];
```

ARGUMENTS:

path

Pointer to a pathname that identifies the new process bound unit.

argv

Array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, argv must have at least one member, and it must point to a string that is the same as path (or its file name component). The array is terminated by a null character pointer.

envp

Array of character pointers to null-terminated strings. These strings constitute the environment for the new process. The array is terminated by a null character pointer.

DESCRIPTION:

The execve function transforms the calling process into a new process. The new process is constructed from an ordinary bound unit called the new process bound unit. There can be no return from a successful exec because the calling process is overlaid by the new process.

When a C program is executed, it is called as follows:

```
int main (argc, argv, envp)
int argc;
unsigned char **argv, **envp;
```

where argc is the argument count and argv is an array of character pointers to the arguments themselves. By convention, argc is at least one and argv[0] points to a string containing the name of the file.

This function allows MOD 400 pathname syntax to be used in the path formal parameter in addition to the UNIX syntax. In fact, MOD 400 and UNIX syntax can be mixed in a given path; for example, <LIST/PROG.L is permitted.

A pointer to the environment of the calling process is placed in the global cell:

```
extern unsigned char **environ;
```

It is used to pass the environment of the calling process to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set. For those file descriptors that remain open, the file currency (read or write) is unchanged.

Signals set to be ignored by the calling process, or set to terminate the calling process, remain so set. Signals set to be caught by the calling process will be set to terminate the new process.

The new process also inherits the following attributes from the calling process:

- Process ID
- Parent process ID
- Process group ID
- TTY group ID
- Time left until an alarm signal
- Current working directory
- Root directory
- File mode creation mask
- File size limit
- Task self-delete switch.

The execve function fails and returns to the calling process if:

- One or more components of the pathname do not exist [ENOENT].

- A directory-name component of path is not a directory [ENOTDIR].

- List access is denied for a directory named in path [EACCES].

- The new process bound unit is not a bound unit, or the calling process lacks execute access to it [EACCES].

- The new process requires more memory than is allowed [ENOMEM].
- The number of characters in the argument list for the new process is greater than 5120 characters [E2BIG].
- The path, argv, or envp argument points to an invalid address [EFAULT].

RETURN VALUE:

If execve returns to the calling process, an error has occurred; the return value is -1, and the variables m4_errno and errno are set to indicate the error.

RELATED FUNCTIONS:

execl, execle, execv, exit, fork, getenv; see also the dl_env, get_env, list_env, and set_env commands.

execlp

Execute a bound unit.

FORMAT:

```
int execlp(file,arg0,arg1,...,argn(unsigned char *)0)
unsigned char *file, *arg0, *arg1, ..., *argn;
```

ARGUMENTS:

file

Pointer to the filename of the new process bound unit.

arg_0, arg_1, ..., arg_n

Pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least arg_0 must be present and point to a string that is the same as path (or its filename component).

DESCRIPTION:

The execlp function transforms the calling process into a new process. The new process is constructed from an ordinary bound unit called the new process bound unit. There can be no return from a successful exec because the calling process is overlaid by the new process.

The directory containing the new process bound unit is found by searching the directories passed as the environment line "PATH= ... ". The PATH environment generated for a program loaded by the MOD 400 command processor specifies the referencing directory, the working directory, >>SYSLIB1, and >>SYSLIB2, in that order.

A pointer to the environment of the calling process is placed in the global cell:

```
extern unsigned char **environ;
```

It is used to pass the environment of the calling process to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see fcntl. For those file descriptors that remain open, the file currency (read or write) is unchanged.

Signals set to be ignored by the calling process, or set to terminate the calling process, remain so set. Signals set to be caught by the calling process are set to terminate the new process.

The new process also inherits the following attributes from the calling process:

- Process ID
- Parent process ID
- Process group ID
- TTY group ID
- Time left until an alarm signal
- Current working directory
- Root directory
- File mode creation mask
- File size limit.

The execlp function fails and returns to the calling process if:

- One or more components of a directory named in the environment line "PATH= ... " does not exist [ENOENT].

- A directory-path component of "PATH= ... " is not a directory [ENOTDIR].

- List access is denied for a directory named in "PATH= ... " [EACCES].

- The new process bound unit is not a bound unit, or the calling process lacks execute access to it [EACCES].

- The new process requires more memory than is allowed [ENOMEM].

- The argv argument points to an invalid address [EFAULT].

RETURN VALUE:

If execlp returns to the calling process, an error has occurred; the return value is -1, and the variables m4_errno and errno are set to indicate the error.

RELATED FUNCTIONS:

execvp, getenv; see also the dl_env, get_env, list_env, and set_env commands.

execvp

Execute a bound unit.

FORMAT:

```
int execvp (file, argv)
unsigned char *file, *argv []
```

ARGUMENTS:

file

Pointer to the filename of the new process bound unit.

argv

Array of character pointers to null-terminated strings. These strings constitute the argument list available to the new task. By convention, argv must have at least one member, and it must point to a string that is the same as path (or its file name component). The array is terminated by a null character pointer.

DESCRIPTION:

The execlp function transforms the calling process into a new process. The new process is constructed from an ordinary bound unit called the new process bound unit. There can be no return from a successful exec because the calling process is overlaid by the new process.

The directory containing the new process bound unit is found by searching the directories passed as the environment line "PATH= ... ". The PATH environment generated for a program loaded by the MOD 400 command processor specifies the referencing directory, the working directory, >>SYSLIB1, and >>SYSLIB2, in that order.

A pointer to the environment of the calling process is placed in the global cell:

```
extern unsigned char **environ;
```

It is used to pass the environment of the calling process to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see fcntl. For those file descriptors that remain open, the file currency (read or write) is unchanged.

Signals set to be ignored by the calling process, or set to terminate the calling process, remain so set. Signals set to be caught by the calling process are set to terminate the new process.

The new process also inherits the following attributes from the calling process:

- Process ID
- Parent process ID
- Process group ID
- TTY group ID
- Time left until an alarm signal
- Current working directory
- Root directory
- File mode creation mask
- File size limit.

The execvp function fails and returns to the calling process if:

- One or more components of a directory named in the environment line "PATH= ... " does not exist [ENOENT].
- A directory-path component of "PATH= ... " is not a directory [ENOTDIR].
- List access is denied for a directory named in "PATH= ... " [EACCES].
- The new process bound unit is not a bound unit, or the calling process lacks execute access to it [EACCES].
- The new process requires more memory than is allowed [ENOMEM].
- The argv argument points to an invalid address [EFAULT].

RETURN VALUE:

If execvp returns to the calling process, an error has occurred; the return value is -1, and the variables m4_errno and errno are set to indicate the error.

RELATED FUNCTIONS:

execlp, getenv; see also the dl_env, get_env, list_env, and set_env commands.

exit

Terminate a process.

FORMAT:

```
exit (status)
int status;
```

ARGUMENTS:

status

Status of operation.

DESCRIPTION:

The exit function terminates the calling process with the following consequences:

- All of the file descriptors open in the child (calling) process are closed.
- If the parent process of the calling process is executing a wait, it is notified of the calling process's termination and the process's content of status is made available to it.
- Any wait timer request of the parent process or alarm request of the child process is stopped and deleted.
- If the process was created by a fork (that is, a process compatible with UNIX and not MOD 400), it is made dormant if its parent process is alive. The child process deletes itself.
- The child process deletes any of its child processes that are dormant. Any of its child processes that are not dormant are made to appear to be children of the UNIX initialization process by changing their parent process ID to one.
- A native MOD 400 process severs relations with its parent without need for further action.

RELATED FUNCTIONS:

signal, wait.

exp

Exponential function.

FORMAT:

```
# include <math.h>

double exp (x)
double x;
```

ARGUMENTS:

x

Double-precision value to be operated on.

DESCRIPTION:

The exp function returns e^x.

DIAGNOSTICS:

The exp function returns a huge value when the correct value would overflow. A very large argument can also result in errno being set to ERANGE.

RELATED FUNCTIONS:

hypot, log, pow, sinh, sqrt.

fabs

Absolute value function.

FORMAT:

```
double fabs (x)
double x;
```

ARGUMENTS:

x

Double-precision value to be operated on.

DESCRIPTION:

The fabs function returns |x| (that is, the absolute value of x).

RELATED FUNCTIONS:

abs, ceil, floor, fmod.

fclose

Close a file.

FORMAT:

```
# include <stdio.h>

int fclose (file)
FILE *file;
```

ARGUMENTS:

file

File pathname.

DESCRIPTION:

The fclose function causes any buffers for the named file to be written to that file, and the file to be closed. Buffers allocated by the standard input/output system are freed.

The fclose function is performed automatically upon calling exit.

RETURN VALUE:

This function returns 0 for success, and EOF if any errors were detected.

RELATED FUNCTIONS:

close, fflush, fopen, setbuf.

fcntl

File control.

FORMAT:

```
# include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd, arg;
```

ARGUMENTS:

fildes

Open file descriptor obtained from a creat, open, dup, or fcntl function.

cmd

Command (see below).

arg

Argument to cmd.

DESCRIPTION:

The fcntl function provides for control over open files.

Acceptable values for cmd are as follows:

F_DUPFD Duplicate the lowest-numbered available file descriptor greater than or equal to arg. The file descriptor shares the same open file(s), file pointer, access mode, and file status flags as the original. The close-on-exec flag associated with the new file descriptor is set to remain open across exec functions.

F_GETFD Get the close-on-exec flag associated with the file descriptor fildes. If the low-order bit is zero, the file remains open across exec functions; otherwise, the file is closed on execution of exec.

F_SETFD Set the close-on-exec flag associated with the file descriptor fildes to the low-order bit of arg.

F_GETFL Get the status flags of file.

F_SETFL Set the status flags of file to arg.

RETURN VALUE:

Upon successful completion, the value returned depends on the cmd argument, as follows:

F_DUPFD --	A new file descriptor
F_GETFD --	Value of flag (only low-order bit defined)
F_SETFD --	Value other than -1
F_GETFL --	Value of file flags
F_SETFL --	Value other than -1.

Otherwise, a value of -1 is returned and the variables errno and m4_errno are set to indicate the error.

DIAGNOSTICS:

The fcntl function fails if:

- The fildes argument does not point to a valid, open file descriptor [EBADF].
- The cmd argument is F_DUPFD and twenty file descriptors are currently open [EMFILE].
- The cmd argument is F_DOPFD and the arg argument is negative or greater than twenty [EINVAL].

RELATED FUNCTIONS:

close, exec, open.

fcvt

Output conversion.

FORMAT:

```
char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;
```

ARGUMENTS:

value

Double-precision value to be converted.

ndigit

Number of digits to be returned.

decpt

Pointer to position of the decimal point relative to the beginning of the string (negative means to the left of the returned digits).

sign

If the sign of the result is negative, the word pointed to by sign is nonzero; zero otherwise.

DESCRIPTION:

The ecvt function converts a value to a null-terminated string of ndigit digits and returns a pointer thereto. The correct digit has been rounded for FORTRAN F-format output of the number of digits specified by the ndigit argument.

NOTE

The return values point to static data whose contents are overwritten by each call.

RELATED FUNCTIONS:

ecvt, gcvt, printf.

fdopen

Open a file.

FORMAT:

```
# include <stdio.h>

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

ARGUMENTS:

fildes

Number of a file descriptor.

type

Access type (see below).

DESCRIPTION:

The fdopen function opens a file descriptor obtained from the open, dup, or creat function. The read/write indicator is set according to the type argument.

When a file is opened for update, both input and output are allowed.

The type argument consists of all valid combinations of r, w, a, +, and b. The argument has these meanings:

```
r   -- Open text file for reading only
w   -- Create text file for writing
a   -- Append to text file
r+  -- Update (read/write) text file
w+  -- Create text file for update (read/write)
a+  -- Append (read/write) at end of text file
rb  -- Open binary file for reading only
wb  -- Open binary file for writing
ab  -- Append to binary file
rb+ -- Update (read/write) binary file
wb+ -- Create binary file for update (read/write)
ab+ -- Append (read/write) at end of binary file.
```

If the file is empty, the type arguments a+ and ab+ are treated as w+ and wb+, respectively.

*

An operation on a text file converts each record to a character stream ending with a newline character, and vice versa. An operation on a binary file transfers fixed-length records directly. In either case, the file is treated as a stream of characters processed by the getc and putc macrocalls. (The buffering required precludes using both getc and putc on a file opened for updating.)

RELATED FUNCTIONS:

fclose, fopen, freopen, open.

feof

File status inquiry -- check for end of file.

FORMAT:

```
# include <stdio.h>

int feof (file)
FILE *file;
```

ARGUMENTS:

file

File pathname.

DESCRIPTION:

The feof function returns nonzero when EOF is read on the named input file; otherwise, it returns zero.

The feof function is a macrocall; it cannot be redeclared.

RELATED FUNCTIONS:

clearerr, ferror, fileno, fopen, open.

ferror

File status inquiry -- check for I/O error.

FORMAT:

```
# include <stdio.h>

int ferror (file)
FILE *file
```

ARGUMENTS:

file

File pathname.

DESCRIPTION:

The ferror function returns a nonzero value when an error has occurred while reading or writing the named file; otherwise, it returns zero. Unless cleared by the clearerr function, the error indication remains until the file is closed.

The ferror function is a macrocall; it cannot be redeclared.

RELATED FUNCTIONS:

clearerr, feof, fileno, fopen, open.

fflush

Flush a file.

FORMAT:

```
# include <stdio.h>

int fflush (file)
FILE *file;
```

ARGUMENTS:

file

File pathname.

DESCRIPTION:

The fflush function causes any buffered data for the named output file to be written to that file.

RETURN VALUE:

This function returns 0 for success, and EOF if any errors were detected.

RELATED FUNCTIONS:

close, fclose, fopen, setbuf.

fgetc

Get character from file.

FORMAT:

```
# include <stdio.h>

int fgetc (file)
FILE *file;
```

ARGUMENTS:

file

File pathname.

DESCRIPTION:

The fgetc function returns the next character from the named input file. The fgetc function behaves like getc, but is a genuine function, not a macrocall; it can therefore be used as an argument. The fgetc macrocall runs more slowly than getc, but takes less space per invocation.

DIAGNOSTICS:

This function returns the value -1 at end of file.

RELATED FUNCTIONS:

ferror, fopen, fread, getc, getchar, gets, getw, putc, scanf.

fgets

Get characters from a file.

FORMAT:

```
# include <stdio.h>

char *fgets (s, n, file)
char *s;
int n;
FILE *file;
```

ARGUMENTS:

s

Pointer to string of characters returned, including a newline character.

n

Number of characters to get -1.

file

File pathname.

DESCRIPTION:

The fgets function reads n-1 characters, or up to a newline character (which is retained), whichever comes first, from the file into the string s. The last character read into s is followed by a null character.

RETURN VALUE:

The fgets function returns its first argument.

DIAGNOSTICS:

The fgets function returns the constant pointer NULL upon the end of file or on an error.

NOTE

The fgets function retains in string s a newline character that ends input.

RELATED FUNCTIONS:

ferror, fopen, fread, getc, gets, puts, scan.

fileno

File status inquiry -- get file descriptor.

FORMAT:

```
# include <stdio.h>

fileno (file)
FILE *file;
```

ARGUMENTS:

file

File pathname.

DESCRIPTION:

The fileno function returns the integer file descriptor associated with the file (see open).

The fileno function is a macrocall; it cannot be redeclared.

RELATED FUNCTIONS:

clearerr, feof, ferror, fopen, open.

find_file

Find a file.

FORMAT:

```
int find_file (file, path [, mode])
unsigned char *file, *path;
[int mode;]
```

ARGUMENTS:

file

Pointer to a null-terminated string naming the file sought.

path

Pointer to a character string of at least 59 characters into which the pathname of the found file is returned. This pathname is terminated by a blank followed by a null.

[mode]

Optional bit pattern giving the access required on the file, constructed as follows:

04 -- Read
02 -- Write
01 -- Execute
00 -- Check file existence only

The default value is 00.

DESCRIPTION:

The directory containing this file is found by searching the directories passed as the environment line "PATH= ... ". The PATH environment generated for a program loaded by the MOD 400 command processor specifies the referencing directory, the working directory, >>SYSLIB1, and >>SYSLIB2, in that order.

You can specify a directory in the "PATH= ..." environment line by giving its absolute or relative pathname or by giving one of the keywords listed below. Relative pathnames are expanded each time the find_file function is invoked. Likewise, the keywords are interpreted by find_file. There is no restriction as to the position of these keywords within the environment line. The acceptable keywords are:

```
-hd  -- Home directory
-rd  -- Referencing directory
-wd  -- Working directory
-sl1 -- >>SYSLIB1
-sl2 -- >>SYSLIB2
```

The find_file function fails if:

- One or more of the components of a directory named in the environment line "PATH= ..." does not exist [ENOENT].
- A component of a directory pathname listed in the environment line "PATH= ... " does not name a directory [ENOTDIR].
- List permission is denied for a directory named in the environment line "PATH= ... " [EACCES].
- The Access Control List on the file denies the requested permission [EACCES].

RETURN VALUE:

If find_file is successful, it returns the value zero. If an error occurs, the return value is -1, and errno and m4_errno are set to indicate the error.

floor

Floor function.

FORMAT:

```
double floor (x)
double x;
```

ARGUMENTS:

x

Double-precision value for comparison.

DESCRIPTION:

The floor function returns the largest integer (as a double-precision number) not greater than x.

RELATED FUNCTIONS:

abs, ceil, fabs, fmod.

fmod

Remainder function.

FORMAT:

```
double fmod (x, y)
double x, y;
```

ARGUMENTS:

x

Double-precision value.

y

Double-precision value.

DESCRIPTION:

The fmod function returns x if y is 0; otherwise, it returns the number f with the same sign as x such that x = i*y + f, for some integer i, and 0 < f < y.

RELATED FUNCTIONS:

abs, ceil, fabs, floor.

fopen

Open a file.

FORMAT:

```
# include <stdio.h>

FILE *fopen (filename, type)
char *filename, *type;
```

ARGUMENTS:

filename

File pathname.

type

Access type (see below).

DESCRIPTION:

The fopen function opens the file named by filename and associates a file with it.

The fopen function returns a file pointer that identifies the file in subsequent operations.

When a file is opened for update, both input and output are allowed.

The type argument consists of all valid combinations of r, w, a, +, and b. The argument has these meanings:

```
r   -- Open text file for reading only
w   -- Create text file for writing
a   -- Append to text file
r+  -- Update (read/write) text file
w+  -- Create text file for update (read/write)
a+  -- Append (read/write) at end of text file
rb  -- Open binary file for reading only
wb  -- Open binary file for writing
ab  -- Append to binary file
rb+ -- Update (read/write) binary file
wb+ -- Create binary file for update (read/write)
ab+ -- Append (read/write) at end of binary file.
```

NOTE

If the file is empty, the type arguments a+ and ab+ are treated as w+ and wb+, respectively.

An operation on a text file converts each record to a character stream ending with a newline character, and vice versa. An operation on a binary file transfers fixed-length records directly. In either case, the file is treated as a stream of characters processed by the getc and putc macrocalls. (The buffering required precludes using both getc and putc on a file opened for updating.)

DIAGNOSTICS:

The fopen function returns a null pointer if the file cannot be accessed.

RELATED FUNCTIONS:

fclose, fdopen, freopen, open.

fork

Create a new process.

FORMAT:

```
int fork ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The fork function creates a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID.
- The child process has its own copy of the parent's file descriptors. Each of the child's open file descriptors shares currency with the corresponding parental file descriptor.

In MOD 400, all users share a single user root directory definition and all tasks in a task group share a single current working directory definition. Thus they are properly inherited by a child process; but neither the child nor parent can subsequently change the working directory without affecting the other and neither can do anything to change the user root directory.

RETURN VALUE:

The fork function returns the process ID of the child to the parent.

DIAGNOSTICS:

The fork function fails and no child process is created if:

- There is not enough group sharable memory available [ENOMEM].
- There is not an available logical resource number [EAGAIN].
- A fork is attempted when running outside of a swappool [ENOSWP].

The fork subroutine can only be used by bound units running in a swappool. If a fork is attempted when running outside a swappool, errno is set to ENOSWP and -1 is returned.

RETURN VALUE:

Upon successful completion, fork returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child is created, and the variables errno and m4_errno are set to indicate the error.

RELATED FUNCTIONS:

exec family, wait.

fprintf

fprintf

Format output to file.

FORMAT:

```
# include <stdio.h>

int fprintf (file, format [, arg] ... )
FILE *file;
char *format;
```

ARGUMENTS:

file

Pathname of file to receive output.

format

Format string (see below).

arg

Optional argument to be printed.

DESCRIPTION:

The fprintf function places output on the named output file. This function converts, formats, and prints its arguments under control of the format. The format is a character string that contains two types of objects: plain characters, which are simply copied to the output file, and conversion specifications, each of which results in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are simply ignored.

Each conversion specification is introduced by the percent (%) character. After the percent character, the following appear in sequence:

- Zero or more flags, which modify the meaning of the conversion specification.

- An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it is blank-padded on the left (or right, if the left-adjustment flag has been given) to make up the field width.

- A precision that gives the minimum number of digits to appear for the d, o, u, x, or X conversions, the number of digits to appear after the decimal point for the e and f conversions, the maximum number of significant digits for the g conversion, the maximum number of characters to be printed from a string in s conversion, or the minimum number of digits to appear in the word address portion of a converted pointer for the p or P conversions. The precision takes the form of a period (.) followed by a decimal digit string; a null digit string is treated as zero.

- An optional l specifying that a following d, o, u, x, or X conversion character applies to a long integer argument.

- A character that indicates the type of conversion to be applied.

A field width or precision can be indicated by an asterisk (*) instead of a digit string. In this case, an integer argument supplies the field width or precision. The argument that is actually converted is not fetched until the conversion letter is seen, so the arguments specifying field width or precision must appear before the argument (if any) to be converted.

The flag characters and their meanings are:

-	The result of the conversion is left-justified within the field.
+	The result of a signed conversion always begins with a sign (+ or -).
blank	If the first character of a signed conversion is not a sign, a blank precedes the result. This implies that if the blank and + flags both appear, the blank flag is ignored. The p and P conversions ignore this flag.

#	The value is to be converted to an "alternate form." For c, d, s, and u conversions, the flag has no effect. For o conversions, it increases the precision to force the first digit of the result to be a zero. For x (X) conversion, a nonzero result will have 0x (0X) preceding it. For e, E, f, g, and G conversions, the result always contains a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For g and G conversions, trailing zeros are not removed from the result (as they normally are). For p or P conversions, the word-address and character-address portions of the converted pointer will each be preceded by 0x or 0X, except when the portion's value is zero.

The conversion characters and their meanings are:

d,o,u,x,X	The integer argument is converted to signed decimal, unsigned octal, unsigned decimal, or unsigned hexadecimal notation (x and X), respectively; the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear, if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a 0 value with a precision of 0 is a null string (unless the conversion is o, x, or X and the # flag is present).
f	The float or double argument is converted to decimal notation in the style "[-]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.
e,E	The float or double argument is converted in the style "[-]d.ddde+dd", where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is 0, no decimal point appears. The E format code produces a

number with E instead of e introducing the exponent. The exponent always contains exactly two digits.

g, G — The float or double argument is printed in style e (or in style E in the case of a G format code), with the precision specifying the number of significant digits. The style used depends on the value converted; style e is used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros are removed from the result; a decimal point appears only if it is followed by a digit.

c — The character argument is printed.

s — The argument is taken to be a string (character pointer) and characters from the string are printed until a null character (\0) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed.

p,P — The pointer argument is printed with the word-address portion of the pointer taken as a long integer converted to unsigned hexadecimal notation, followed immediately by the character-offset portion taken as an integer converted to signed hexadecimal notation and enclosed within parentheses. The letters abcdef are used for the digits greater than nine in the p conversion. The letters ABCDEF are used for the digits greater than nine in the P conversion. The precision specifies the minimum number of digits to appear in the converted word-address portion of the pointer. If the converted value can be represented with fewer digits, it is expanded with leading zeros. The precision has no effect on the conversion of the character-offset portion of the pointer. The default precision is one. An explicit precision of zero is treated as if one was specified.

% — Print a %; no argument is converted.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by fprintf are printed as if putchar had been called.

RETURN VALUE:

This function returns the number of characters transmitted.

DIAGNOSTICS:

If this function encounters an invalid string pointer, it behaves as if it has encountered a valid pointer to a null string. An error condition is indicated to the calling function by a negative return value.

EXAMPLES:

To print a date and time in the form "Sunday, July 3, 10:02", where weekday and month are pointers to null-terminated strings:

```
fprintf(temp,"%s, %s %d, %.2d:%.2d",weekday,month,day,hour,min);
```

To print pi to five decimal places:

```
fprintf(output,"pi = %.5f", 4*atan(1.0));
```

RELATED FUNCTIONS:

ecvt, printf, putc, scanf, sprintf.

fputc

Put a character on a file.

FORMAT:

```
# include <stdio.h>

fputc (c, file)
FILE *file;
```

ARGUMENTS:

c

Character to write to file.

file

File pathname.

DESCRIPTION:

The fputc function appends the character c to the named output file. Unlike putc, it is a genuine function rather than a macrocall; it can therefore be used as an argument. The fputc function runs more slowly than putc, but takes less space per invocation.

RETURN VALUE:

The fputc function returns the character written.

DIAGNOSTICS:

The fputc function returns the constant EOF when it encounters an error. Since this is a good integer, ferror should be used to detect putw errors.

RELATED FUNCTIONS:

ferror, fopen, fwrite, getc, printf, putc, putchar, puts, putw.

fputs

Put a string on a file.

FORMAT:

```
# include <stdio.h>

int fputs (s, file)
char *s;
FILE *file;
```

ARGUMENTS:

s

String to be written to the file.

file

File pathname.

DESCRIPTION:

The fputs function copies the null-terminated string s to the named output file.

This function does not copy the terminating null character.

DIAGNOSTICS:

This function returns EOF if it encounters an error.

NOTE

The fputs function does not append a newline character.

RELATED FUNCTIONS:

ferror, fopen, fwrite, gets, printf, putc, puts.

fread

Buffered input.

FORMAT:

```
# include <stdio.h>

fread (buf_ptr, size, nitems, file)
int size;
int nitems;
char *buf_ptr;
FILE *file;
```

ARGUMENTS:

buf_ptr

Buffer address pointer.

size

Item size in characters.

nitems

Number of items to read.

file

File pathname.

DESCRIPTION:

The fread function reads, into an array beginning at buf_ptr, nitems of size characters each from the named input file.

RETURN VALUE:

The fread function returns the number of items actually read.

RELATED FUNCTIONS:

fopen, fwrite, getc, gets, printf, putc, puts, read, scanf, write.

free

Free heap memory.

FORMAT:

```
void free (ptr)
char *ptr;
```

ARGUMENTS:

ptr

Pointer to a block previously allocated by calloc or malloc; this space is made available for further allocation.

DESCRIPTION:

The malloc and free functions together provide a simple, general-purpose memory allocation package.

DIAGNOSTICS:

Unspecified results occur if free acts on some random number.

RELATED FUNCTIONS:

calloc, malloc, realloc.

freopen

Reopen a file.

FORMAT:

```
# include <stdio.h>

FILE *freopen (filename, type, file)
char *filename, *type;
FILE *file;
```

ARGUMENTS:

filename

New file pathname.

type

Access type (see below).

file

Old file pathname.

DESCRIPTION:

The freopen function substitutes the named file in place of the open file. It returns the original value of file. The original file is closed, regardless of whether the open ultimately succeeds.

The freopen function is used to attach the pre-opened constant names stdin, stdout, and stderr to specified files.

When a file is opened for update, both input and output are allowed.

The type argument consists of all valid combinations of r, w, a, +, and b. The argument has these meanings:

```
r   -- Open text file for reading only
w   -- Create text file for writing
a   -- Append to text file
r+  -- Update (read/write) text file
w+  -- Create text file for update (read/write)
a+  -- Append (read/write) at end of text file
rb  -- Open binary file for reading only
wb  -- Open binary file for writing
ab  -- Append to binary file
```

rb+ -- Update (read/write) binary file
wb+ -- Create binary file for update (read/write)
ab+ -- Append (read/write) at end of binary file.

If the file is open, the type arguments a+ and ab+ are treated as w+ and wb+, respectively.

An operation on a text file converts each record to a character stream ending with a newline character, and vice versa. An operation on a binary file transfers fixed-length records directly. In either case, the file is treated as a stream of characters processed by the getc and putc macrocalls. (The buffering required precludes using both getc and putc on a file opened for updating.)

DIAGNOSTICS:

The freopen function returns a null pointer if filename cannot be accessed.

RELATED FUNCTIONS:

fclose, fdopen, fopen, open.

frexp

Split into mantissa and exponent.

FORMAT:

```
double frexp (value, eptr)
double value;
int *eptr;
```

ARGUMENTS:

value

Double-precision value to be processed.

eptr

Pointer to exponent.

DESCRIPTION:

The frexp function returns the mantissa, x, of the double-precision value as a double-precision quantity. The magnitude of x is less than 1 and greater than 1/16. It stores the exponent at the location pointed to by eptr. The exponent is the integer n such that value = $x*2^n$.

RELATED FUNCTIONS:

ldexp, modf.

fscanf

Formatted input conversion.

FORMAT:

```
# include <stdio.h>

fscanf (file, format [, pointer]...)
FILE *file;
char *format;
```

ARGUMENTS:

file

Input file pathname.

format

Control string format (see below).

pointer

Set of arguments indicating where the converted input should be stored.

DESCRIPTION:

The fscanf function reads from the named input file. This function reads characters, interprets them according to a format, and stores the results in its arguments. It requires a control string format described below, and an optional set of pointer arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs, or newline characters, which cause input to be read up to the next non-white-space character.

2. An ordinary character (not %), which must match the next character of the input file.

3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. An input field is defined as a string of nonspace characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are valid:

% A single % is expected in the input at this point; no assignment is done.

d A decimal integer is expected; the corresponding argument should be an integer pointer.

o An octal integer is expected; the corresponding argument should be an integer pointer.

x A hexadecimal integer is expected; the corresponding argument should be an integer pointer.

s A character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0, which is added automatically. The input field is terminated by a space or newline character.

c A character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next nonspace character, use %1s. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

e,f A floating-point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float. The input format for floating-point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an E or an e, followed by an optionally signed integer.

[Indicates a string that is not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not a circumflex (^), the input field consists of all characters up to the first character that is not in the set between the brackets; if the first character after the left bracket is a circumflex, the input field consists of all characters up to the first character that is in the set of the remaining characters between the brackets. The corresponding argument must point to a character array.

The conversion characters d, o, and x can be capitalized and/or preceded by l to indicate that a pointer to long rather than to int is in the argument list. Similarly, the conversion characters e and f may be capitalized and/or preceded by l to indicate that a pointer to double rather than to float is in the argument list.

The fscanf conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input file.

RETURN VALUE:

The fscanf function returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

NOTE

Trailing white space (including a newline character) is left unread unless matched in the control string.

DIAGNOSTICS:

This function returns EOF at the end of input and a short count for missing or illegal data items.

NOTE

The success of literal matches and suppressed assignments is not directly determinable.

EXAMPLES:

The call:

```
int i; float x; char name[50];
fscanf (names, "%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 brenda
```

assigns to i the value 25, to x the value 5.432, and name contains brenda\0. Or:

```
int i; float x; char name[50];
fscanf (data, "%2d%f%*d%[1234567890]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

assigns 56 to i, 789.0 to x, skip 0123, and places the string 56\0 in name. The next call to getchar returns a.

RELATED FUNCTIONS:

atof, getc, printf, scanf, sscanf.

fstat

Get file status.

FORMAT:

```
# include <types.h>
# include <stat.h>

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

ARGUMENTS:

fildes

File descriptor of the open file.

buf

Pointer to a static structure into which information is placed concerning the file.

DESCRIPTION:

The fstat function obtains information about an open file known by the file descriptor fildes, obtained from a successful open, creat, or dup function.

The contents of the structure pointed to by buf include the following members:

```
ushort   st_mode;   /*File mode                            */
ino_t    st_ino;    /*Inode number (N/A in MOD 400)        */
dev_t    st_dev;    /*ID of device containing              */
                    /*a directory entry for this file      */
dev_t    st_rdev;   /*ID of device                         */
                    /*This entry is defined only for       */
                    /*character special or block special
                      files                                */
short    st_nlink;  /*Number of links (N/A in MOD 400)     */
ushort   st_uid;    /*User ID of the file's owner          */
ushort   st_gid;    /*Group ID of the file's group         */
off_t    st_size;   /*File size in characters (N/A)        */
time_t   st_atime;  /*Time of last access                  */
time_t   st_mtime;  /*Time of last data modification       */
                    /*Times measured in seconds since
                      00:00:00 GMT, Jan. 1, 1970           */
```

The st_atime member is the date/time when the file was last accessed. It is changed by the functions creat, pipe, and read.

The st_mtime member is the date/time when the file was last modified. It is changed by the functions creat, pipe, and write.

The st_ctime member is the date/time when the file was created. It is changed by the functions creat, link, pipe, unlink, and write.

Information is not available in the members st_ino, st_nlink, and st_size.

The fstat function fails if:

- The fildes argument is not a valid open file descriptor [EBADF].
- The buf argument points to an invalid address [EFAULT].

RETURN VALUE:

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and errno and m4_errno are set to indicate the error.

RELATED FUNCTIONS:

creat, link, stat, time, unlink.

fwrite

Buffered output.

FORMAT:

```
# include <stdio.h>

fwrite (buf_ptr, size, nitems, file)
int size;
int nitems;
char *buf_ptr;
FILE *file;
```

ARGUMENTS:

buf_ptr

Buffer address pointer.

size

Item size in characters.

nitems

Number of items to write.

file

File pathname.

DESCRIPTION:

The fwrite function appends at most nitems of size size beginning at buf_ptr to the named output file. It returns the number of items actually written.

RELATED FUNCTIONS:

fopen, fread, gets, printf, putc, puts, read, scanf, write.

gamma

Log gamma function.

FORMAT:

```
# include <math.h>
extern int signgam;

double gamma (x)
double x;
```

ARGUMENTS:

x

Double-precision positive value to be processed.

signgam

Returned sign of gamma function.

DESCRIPTION:

The gamma (x) function computes the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the external variable signgam. X must be a positive value.

The gamma function is defined as:

$$(x) = \quad \exp(-t)\ t^{x-1}\ dt$$

DIAGNOSTICS:

For nonpositive integer arguments, a huge value (HUGE) is returned, and the variable errno is set to EDOM.

gcvt

Output conversion.

FORMAT:

```
char *gcvt (value, ndigit, buf)
double value;
int ndigit;
char *buf;
```

ARGUMENTS:

value

Value to be converted.

ndigit

Number of significant digits.

buf

Pointer to output string.

DESCRIPTION:

The gcvt function converts the argument value to a null-terminated string pointed to by buf and returns buf. It attempts to produce ndigit significant digits in FORTRAN F-format if possible; otherwise it produces output in E-format, ready for printing. Trailing zeros are suppressed.

NOTE

The return values point to static data whose contents are overwritten by each call.

RELATED FUNCTIONS:

ecvt, fcvt, printf.

getc

Get character from file.

FORMAT:

```
# include <stdio.h>

int getc (file)
FILE *file;
```

ARGUMENTS:

file

File pathname.

DESCRIPTION:

The getc function returns the next character from the buffer associated with the named input file. The function obtains a new buffer's worth of characters whenever all the characters have been returned..

DIAGNOSTICS:

This function returns the value -1 when it encounters the end of a file.

NOTE

Because it is a macrocall, getc treats incorrectly a file argument with side effects; for example:

```
getc(*f++);
```

RELATED FUNCTIONS:

ferror, fgetc, fopen, fread, getchar, gets, getw, putc, scanf.

getchar

Get character from stdin file.

FORMAT:

```
# include <stdio.h>

int getchar ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The getchar function is identical to getc(stdin). This function is implemented as a macrocall; it cannot be redefined.

DIAGNOSTICS:

This function returns the value -1 when it encounters the end of a file.

RELATED FUNCTIONS:

ferror, fgetc, fopen, fread, getc, gets, getw, putc, scanf.

getcwd

Get current working directory.

FORMAT:

```
char *getcwd (buf, size)
char *buf;
int size;
```

ARGUMENTS:

buf

Returned current working directory string.

size

Buffer size in characters.

DESCRIPTION:

The getcwd function returns a pointer to the null-terminated character string of the current working directory.

The value of the size argument must be at least one character longer than the pathname to be returned. Under MOD 400, the maximum length of a directory path is 44 characters.

If the buf argument is a null pointer, getcwd obtains size characters of space using the malloc function. In this case, you can use the returned pointer in a subsequent call to the free function.

If the buf argument is not a null pointer, the string is placed in buf, and the pointer to buf is returned.

DIAGNOSTICS:

If an error occurs, a null pointer is returned.

getdir

Get pathname of a system directory.

FORMAT:

```
unsigned char *getdir (buf, dir)
unsigned char *buf;
int dir;
```

ARGUMENTS:

buf

Pathname of returned directory.

dir

Specifies directory whose pathname is to be returned:

-2 -- Referencing directory
-1 -- Home directory
 0 -- Working directory
+1 -- >>SYSLIB1
+2 -- >>SYSLIB2.

DESCRIPTION:

The getdir function stores the pathname of the specified system directory in buf and returns buf plus the length of the stored pathname (not including the terminating null).

RETURN VALUE:

If the getdir function is successful, it returns a pointer to the null character terminating the pathname stored in buf. If it is unsuccessful, errno and m4_errno are set to indicate the error and (unsigned char *) 0 is returned.

getegid

Get effective group ID.

FORMAT:

```
int getegid ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The getegid function returns the effective group ID of the calling process. The sum of the characters of the MOD 400 account ID is used as the effective group ID.

RELATED FUNCTIONS:

getuid, geteuid, getgid.

getenv

Get environment name.

FORMAT:

```
char *getenv (name)
char *name;
```

ARGUMENTS:

name

Environment name.

DESCRIPTION:

The getenv function searches the environment list for a string of the form name and returns a pointer to that value if such a string is present; otherwise, it returns a null pointer.

geteuid

Get effective user ID.

FORMAT:

```
int geteuid ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The geteuid function returns the effective user ID of the calling process. The MOD 400 task group ID is used for this identifier.

RELATED FUNCTIONS:

getuid, getgid, getegid.

getgid

Get real group ID.

FORMAT:

```
int getgid ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The getgid function returns the real group ID of the calling process. The sum of the characters of the MOD 400 account ID is used as the group ID.

RELATED FUNCTIONS:

getuid, geteuid, getegid.

getgrent

Get group record entry.

FORMAT:

```
# include <grp.h>

struct group *getgrent ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The getgrent function returns a pointer to a static object of the type struct group as defined in the grp.h header file. The MOD 400 account ID is used for the group name, and the sum of its characters as the group ID; a null string is used as the encrypted password; and the caller is listed as the only member of the group.

When first called, the getgrent function returns a pointer to the group structure as described above; thereafter, it returns a null pointer.

This causes the caller to perceive the system as a single-user UNIX system with only one group defined.

RELATED FUNCTIONS:

endgrent, getgrgid, getgrnam, getlogin, getpwent, group, setgrent.

getgrgid

Get group record by group ID.

FORMAT:

```
# include <grp.h>

struct group *getgrgid (gid)
int gid;
```

ARGUMENTS:

gid

Group ID.

DESCRIPTION:

The getgrgid function returns a pointer to a static object of the type struct group as defined in the grp.h header file. The MOD 400 account ID is used for the group name, and the sum of its characters as the group ID; a null string is used as the encrypted password; and the caller is listed as the only member of the group.

This causes the caller to perceive the system as a single-user UNIX system with only one group defined.

DIAGNOSTICS:

This function returns a null pointer if the group ID given as the argument is not the group ID of the caller.

RELATED FUNCTIONS:

endgrent, getgrent, getgrnam, getlogin, getpwent, group, setgrent.

getgrnam

Get group record by group name.

FORMAT:

```
# include <grp.h>

struct group *getgrnam (name)
char *name;
```

ARGUMENTS:

name

Group name.

DESCRIPTION:

The getgrnam function returns a pointer to a static object of the type struct group as defined in the grp.h header file. The MOD 400 account ID is used for the group name, and the sum of its characters as the group ID; a null string is used as the encrypted password; and the caller is listed as the only member of the group.

This causes the caller to perceive the system as a single-user UNIX system with only one group defined.

DIAGNOSTICS:

This function returns a null pointer if the group name given as the argument is not the group name of the caller.

RELATED FUNCTIONS:

endgrent, getgrent, getgrgid, getlogin, getpwent, group, setgrent.

getlogin

Get login name.

FORMAT:

```
char *getlogin ( );
```

ARGUMENTS:

None.

DESCRIPTION:

The getlogin function returns a pointer to a static string containing the login name of the calling process (the MOD 400 person ID). It can be used in conjunction with getpwnam to locate the correct password file entry when the same user ID is shared by several login names.

If getlogin is called within a process that is not attached to a typewriter, it returns (char *) 0 (a null pointer). The correct procedure for determining the login name is to call getlogin; if it fails, call getpwent.

DIAGNOSTICS:

This function returns a null pointer if the name is not found.

NOTE

The return values point to static data whose contents are overwritten by each call.

RELATED FUNCTIONS:

getgrent, getpwent.

getopt

Get option letter from argument.

FORMAT:

```
int getopt (argc, argv, optstring)
int argc;
char **argv;
char *optstring;
extern char *optarg;
extern int optind;
```

ARGUMENTS:

argc

Index into *argv.

argv

Input string of options.

optstring

String of valid options.

DESCRIPTION:

The getopt function returns the next option letter in argv that matches a letter in optstring. The argument optstring is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. The pointer optarg is set to point to the start of the option argument on return from getopt.

The getopt function places in optind the argv index of the next argument to be processed. Because optind is external, it is normally initialized to zero automatically before the first call to getopt.

RETURN VALUE:

When all options have been processed (that is, up to the first nonoption argument), getopt returns EOF. The special option minus (-) can be used to delimit the end of the options; EOF is returned, and minus (-) is skipped.

DIAGNOSTICS:

The getopt function displays an error message and returns a question mark (?) when it encounters an option letter not included in optstring.

EXAMPLE:

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options a and b, and the options f and e, both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
     int c;
     extern int optind;
     extern char *optarg;
     .
     .
     .
     while ((c = getopt (argc, argv, "abf:o:")) != EOF)
     switch (c) {

        case "a":

            if (bfg)
                    errfg++;
            else
                    afg++;
            break;
        case "b":

            if (afg)
                    errfg++;
            else
                    bproc();
            break;
        case "f":
            ifile = optarg;
            break;
        case "o";
            ofile = optarg;
            bufsize = 512;
            break;
        case "?":
            errfg++;
     }
   if (errfg) {
        fprintf (stderr, "usage:...");
        exit;
   }
   for ( ; optind < argc; optind++) {
          if (access (argv[optind], 4)) {
          .
          .
          .
          }
   .
   .
   .
   }
}
```

getpgrp

Get process group ID.

FORMAT:

```
int getpgrp ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The getpgrp function returns the process group ID of the calling process.

RELATED FUNCTIONS:

getpid, getppid.

getpid

Get process ID.

FORMAT:

```
getpid ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The getpid function returns the process ID of the calling process.

This function is useful for generating uniquely named temporary files.

RELATED FUNCTIONS:

getpgrp, getppid.

getppid

Get parent process ID.

FORMAT:

```
getppid ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The getppid function returns the parent (creating) process ID if the calling process was created by fork or one of the run functions and the parent process is still alive. Otherwise the parent process is reported to be process 1 (corresponding to the initialization process of UNIX).

If the calling task was created by the fork subroutine or one of the run functions and the creating task still exists, the creating task's process ID (task control block address shifted five bits right) is returned. In all other cases, the calling task is reported to be a child of process 1 (the initialization process in UNIX).

RELATED FUNCTIONS:

getpid, getpgrp, getptcb.

getptcb

Get parent TCB pointer.

FORMAT:

```
int *getptcb ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The getptcb function returns an integer pointer to the task control block of the calling process's parent process (if its parent is known). The parent process is known only if the calling process was created by the fork function or one of the run functions, and the parent process has not yet terminated. Otherwise a null integer pointer is returned.

RELATED FUNCTIONS:

fork, gettcb, getppid, run family.

getpwent

Get password record entry.

FORMAT:

```
# include <pwd.h>

struct passwd *getpwent ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The getpwent function returns a pointer to a static structure as defined in the pwd.h header file. The MOD 400 person ID is used as the login name. A null string is given as the encrypted password. The MOD 400 task group ID is given as the user ID. The sum of the characters in the MOD 400 account ID is given as the group ID. Null strings are given for the password age, comment, and gecos strings. The task group's home directory is given as the process initial working directory. A null string is given as the name of the (UNIX) shell program.

Subsequent calls to getpwent return a null pointer, unless reinitialized by setpwent.

The effect is to cause the caller to perceive the system as a single-user UNIX system.

RELATED FUNCTIONS:

endpwent, getpwnam, getpwuid, setpwent.

getpwnam

Get password record by login name.

FORMAT:

```
# include <pwd.h>

struct passwd *getpwnam (name)
char *name;
```

ARGUMENTS:

name

Login name.

DESCRIPTION:

The getpwnam function returns a pointer to a static structure as defined in the pwd.h header file. The MOD 400 person ID is used as the login name. A null string is given as the encrypted password. The MOD 400 task group ID is given as the user ID. The sum of the characters in the MOD 400 account ID is given as the group ID. Null strings are given for the password age, comment, and gecos strings. The task group's home directory is given as the process initial working directory. A null string is given as the name of the (UNIX) shell program.

The effect is to cause the caller to perceive the system as a single-user UNIX system.

DIAGNOSTICS

This function returns a null pointer if the login name argument is not the login name of the caller.

RELATED FUNCTIONS:

endpwent, getpwent, getpwuid, setpwent.

getpwuid

Get password record by user ID.

FORMAT:

```
# include <pwd.h>

struct passwd *getpwuid (uid)
int uid;
```

ARGUMENTS:

uid

User ID.

DESCRIPTION:

The getpwuid function returns a pointer to a static structure as defined in the pwd.h header file. The MOD 400 person ID is used as the login name. A null string is given as the encrypted password. The MOD 400 task group ID is given as the user ID. The sum of the characters in the MOD 400 account ID is given as the group ID. Null strings are given for the password age, comment, and gecos strings. The task group's home directory is given as the process initial working directory. A null string is given as the name of the (UNIX) shell program.

The effect is to cause the caller to perceive the system as a single-user UNIX system.

RELATED FUNCTIONS:

endpwent, getpwent, getpwnam, setpwent.

getr

Get record.

FORMAT:

```
# include <ufas.h>

int getr(cmd,fildes,rptr,rlen,rtype,[keyptr,keytype])
int cmd, fildes, rlen, rtype[, keytype];
char *rptr;
record_key *keyptr;
```

ARGUMENTS:

cmd

Command (see "Description").

fildes

Open file descriptor obtained from a creat, open, dup, or fcntl function.

rptr

Pointer to a record area into which the record is to be read.

rlen

Number of characters to read.

rtype

Input record type: -1 if any type is acceptable, or a decimal value from 0 through 3999.

keyptr

Pointer to key value.

keytype

Optional key type:

PRIMARY -- For indexed files; keyvalue type is (char *)

RELATIVE -- For relative files; keyvalue type is (long *)

CALC -- For random files; keyvalue type is (char *)

SIMPLE -- For sequential or dynamic files; keyvalue type is (long *)

ALT -- For alternate index; keyvalue type is (char *)

CURRENT -- Current key of usage.

DESCRIPTION:

The getr function reads a single record from a MOD 400 data management file into memory, according to a key value.

Acceptable values for cmd are as follows:

RD_NXT -- Read next record
RD_KEY -- Read keyed record
RD_DUP -- Read random file with key.

The last two arguments (keyptr and keytype) are optional for sequential read operations.

The record_key data type is defined in the ufas.h file as:

```
typedef union {
            unsigned long n;
            unsigned char s[];
             } record_key;
```

The member n (32 bits) is used for simple or relative keys. The member s (variable) is used for other keys.

A simple key is constructed from the control-interval number and line number of a record according to this formula:

key = (256 * CI) + (line)

For a relative key, n is the value directly. For a primary or CALC key, s is the key value directly. Always specify an alternate key where appropriate.

RETURN VALUE:

Upon successful completion, a value of 0 is returned. Otherwise, the variables errno and m4_errno are set to indicate the error, and a value of -1 is returned.

RELATED FUNCTIONS:

posr, putr.

gets

Get string from stdin file.

FORMAT:

```
# include <stdio.h>

char *gets (s)
char *s;
```

ARGUMENTS:

s

Pointer to buffer that will hold string.

DESCRIPTION:

The gets function reads a string into s from the standard input file stdin. The string is terminated by a newline character, which is replaced in s by a null character. The gets function returns its argument.

DIAGNOSTICS:

The gets function returns a null pointer if it encounters the end of a file or an error.

NOTE

The gets function deletes the newline character ending its input.

RELATED FUNCTIONS:

ferror, fgets, fopen, fread, getc, puts, scan.

gettcb

gettcb

Get TCB pointer.

FORMAT:

```
int *gettcb ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The gettcb function returns an integer pointer to the task control block of the calling process.

RELATED FUNCTIONS:

fork, getpid, getptcb, run family.

getuid

Get real user ID.

FORMAT:

```
int getuid ( )
```

ARGUMENTS:

None.

DESCRIPTION:

The getuid function returns the real user ID of the calling process. The MOD 400 task group ID is used for this identifier.

RELATED FUNCTIONS:

geteuid, getgid, getegid.

getw

Get word from file.

FORMAT:

```
# include <stdio.h>

int getw (file)
FILE *file;
```

ARGUMENTS:

file

File pathname.

DESCRIPTION:

The getw function returns the next word from the named input file. It returns the constant EOF when it encounters the end of a file or an error, but since that is a valid integer value, feof and ferror should be used to check the success of getw. The getw function assumes no special alignment in the file.

DIAGNOSTICS:

This function returns the value -1 when it encounters the end of a file.

RELATED FUNCTIONS:

feof, ferror, fgetc, fopen, fread, getc, getchar, gets, putc, putw, scanf.

gmtime

Convert date and time to ASCII.

FORMAT:

```
struct tm *gmtime (clock)
long *clock;
```

ARGUMENTS:

clock

Military time.

DESCRIPTION:

The gmtime function returns a pointer to a structure containing the components of the time. The gmtime function converts directly to Greenwich Mean Time (GMT).

The structure declaration from the include file is:

```
struct tm   {
        int     tm_sec;
        int     tm_min;
        int     tm_hour;
        int     tm_mday;
        int     tm_mon;
        int     tm_year;
        int     tm_wday;
        int     tm_yday;
        int     tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday - 0), year - 1900, day of year (0-365), and a flag that is nonzero if daylight saving time is in effect.

The external long variable timezone contains the difference, in seconds, between GMT and local standard time (in EST, timezone is 5*60*60); the external variable daylight is nonzero if, and only if, the standard U.S. daylight savings time conversion should be applied.

NOTE

The return values point to static data whose contents are overwritten by each call.

*

RELATED FUNCTIONS:

asctime, ctime, localtime, time, tzset; see also the list_stz and set_stz commands.

hypot

Euclidean distance.

FORMAT:

```
# include <math.h>

double hypot (x, y)
double x, y;
```

ARGUMENTS:

x

Double-precision value.

y

Double-precision value.

DESCRIPTION:

The hypot function returns

$$(x^2 + y^2)$$

taking precautions against unwarranted overflows.

RELATED FUNCTIONS:

sqrt.

init_mem

Initialize memory.

FORMAT:

```
void init_mem (ch_ptr, char_count, fill_char)
```

ARGUMENTS:

ch_ptr

Pointer to the starting location.

char_count

Number of characters to be initialized.

fill_char

Character used to initialize memory.

DESCRIPTION:

This is an obsolete function. It is provided only to maintain compatibility with past releases.

The init_mem function initializes a block of memory to the specified value.

isalnum

Character classification (alphanumeric).

FORMAT:

```
# include <ctype.h>

int isalnum (c)
int c;
```

ARGUMENTS:

c

Single-character value.

DESCRIPTION:

The isalnum macrocall classifies ASCII-coded integer values by table lookup. The macrocall is a predicate returning nonzero for true, zero for false. The isalnum function is defined only where isascii8 is true and on the single non-ASCII value EOF (see isascii8). The function is nonzero if c is an alphanumeric (letter or digit).

RELATED FUNCTIONS:

isalpha, isascii, isascii8, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit.

isalpha

Character classification (alphabetic).

FORMAT:

```
# include <ctype.h>

int isalpha (c)
int c;
```

ARGUMENTS:

c

Single-character value.

DESCRIPTION:

The isalpha macrocall classifies ASCII-coded integer values by table lookup. The macrocall is a predicate returning nonzero for true, zero for false. The isalpha function is defined only where isascii8 is true and on the single non-ASCII value EOF. The function is nonzero if c is a letter.

RELATED FUNCTIONS:

isalnum, isascii, isascii8, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit.

isascii

Character classification (7-bit ASCII).

FORMAT:

```
# include <ctype.h>

int isascii (c)
int c;
```

ARGUMENTS:

c

Single-character value.

DESCRIPTION:

The isascii macrocall classifies 7-bit ASCII-coded integer values by table lookup. The macrocall is a predicate returning nonzero for true, zero for false. The isascii function is defined on all integer values. The function is nonzero if c is a 7-bit ASCII character, that is, a nonnegative integer less than hexadecimal 80.

RELATED FUNCTIONS:

isalnum, isalpha, isascii8, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit.

isascii8

Character classification (8-bit ASCII).

FORMAT:

```
# include <ctype.h>

int isascii8 (c)
int c;
```

ARGUMENTS:

c

Single-character value.

DESCRIPTION:

The isascii8 macrocall classifies 8-bit ASCII-coded integer values by table lookup. The macrocall is a predicate returning nonzero for true, zero for false. The isascii8 function is defined on all integer values. The function is nonzero if c is an ASCII character, code less than hexadecimal 100.

RELATED FUNCTIONS:

isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit.

isatty

Determine if association is to a terminal.

FORMAT:

```
int isatty (fildes)

int fildes;
```

ARGUMENTS:

fildes

File descriptor.

DESCRIPTION:

The isatty function returns 1 if fildes is associated with a terminal device; otherwise, it returns a 0.

iscntrl

Character classification (control character).

FORMAT:

```
# include <ctype.h>

int iscntrl (c)
int c;
```

ARGUMENTS:

c

Single-character value.

DESCRIPTION:

The iscntrl macrocall classifies ASCII-coded integer values by table lookup. The macrocall is a predicate returning nonzero for true, zero for false. The iscntrl function is defined only where isascii8 is true and on the single non-ASCII value EOF. The function is nonzero if c is a delete character (hexadecimal 7F) or ordinary control character (hexadecimal 0 through 17, 84 through 97, and 9B through 9F).

RELATED FUNCTIONS:

isalnum, isalpha, isascii, isascii8, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit.

isdigit

Character classification (digit).

FORMAT:

```
# include <ctype.h>

int isdigit (c)
int c;
```

ARGUMENTS:

c

Single-character value.

DESCRIPTION:

The isdigit macrocall classifies ASCII-coded integer values by table lookup. The macrocall is a predicate returning nonzero for true, zero for false. The isdigit function is defined only where isascii8 is true and on the single non-ASCII value EOF. The function is nonzero if c is a digit [0 through 9].

RELATED FUNCTIONS:

isalnum, isalpha, isascii, isascii8, iscntrl, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit.

isgraph

Character classification (nonspace printing character).

FORMAT:

```
# include <ctype.h>

int isgraph (c)
int c;
```

ARGUMENTS:

c

Single-character value.

DESCRIPTION:

The isgraph macrocall classifies ASCII-coded integer values by table lookup. The macrocall is a predicate returning nonzero for true, zero for false. The isgraph function is defined only where isascii8 is true and on the single non-ASCII value EOF. The function is nonzero if c is a printing character.

RELATED FUNCTIONS:

isalnum, isalpha, isascii, isascii8, iscntrl, isdigit, islower, isprint, ispunct, isspace, isupper, isxdigit.

islower

Character classification (lowercase alphabetic).

FORMAT:

```
# include <ctype.h>

int islower (c)
int c;
```

ARGUMENTS:

c

Single-character value.

DESCRIPTION:

The islower macrocall classifies ASCII-coded integer values by table lookup. The macrocall is a predicate returning nonzero for true, zero for false. The islower function is defined only where isascii8 is true and on the single non-ASCII value EOF. The function is nonzero if c is a lowercase letter. The lowercase letters are hexadecimal 61 through 7A, E0 through F6, and F8 through FF.

RELATED FUNCTIONS:

isalnum, isalpha, isascii, isascii8, iscntrl, isdigit, isgraph, isprint, ispunct, isspace, isupper, isxdigit.

isprint

Character classification (printing character).

FORMAT:

```
# include <ctype.h>

int isprint (c)
int c;
```

ARGUMENTS:

c

Single-character value.

DESCRIPTION:

The isprint macrocall classifies ASCII-coded integer values by table lookup. The macrocall is a predicate returning nonzero for true, zero for false. The isprint function is defined only where isascii8 is true and on the single non-ASCII value EOF. The function is nonzero if c is a printing character; that is, hexadecimal 20 (space) through 7E (tilde), or hexadecimal A0 (no-break space) through FF
* (small letter y with diaeresis).

RELATED FUNCTIONS:

isalnum, isalpha, isascii, isascii8, iscntrl, isdigit, isgraph, islower, ispunct, isspace, isupper, isxdigit.

ispunct

Character classification (punctuation character).

FORMAT:

```
# include <ctype.h>

int ispunct (c)
int c;
```

ARGUMENTS:

c

Single-character value.

DESCRIPTION:

The ispunct macrocall classifies ASCII-coded integer values by table lookup. The macrocall is a predicate returning nonzero for true, zero for false. The ispunct function is defined only where isascii8 is true and on the single non-ASCII value EOF. The function is nonzero if c is a punctuation character (neither control nor alphanumeric).

RELATED FUNCTIONS:

isalnum, isalpha, isascii, isascii8, iscntrl, isdigit, isgraph, islower, isprint, isspace, isupper, isxdigit.

isspace

Character classification (whitespace character).

FORMAT:

```
# include <ctype.h>

int isspace (c)
int c;
```

ARGUMENTS:

c

Single-character value.

DESCRIPTION:

The isspace macrocall classifies ASCII-coded integer values by table lookup. The macrocall is a predicate returning nonzero for true, zero for false. The isspace function is defined only where isascii8 is true and on the single non-ASCII value EOF. The function is nonzero if c is a space, tab, carriage return, newline character, vertical tab, formfeed, or no-break space.

RELATED FUNCTIONS:

isalnum, isalpha, isascii, isascii8, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isupper, isxdigit.

isupper

Character classification (uppercase alphabetic).

FORMAT:

```
# include <ctype.h>

int isupper (c)
int c;
```

ARGUMENTS:

c

Single-character value.

DESCRIPTION:

The isupper macrocall classifies ASCII-coded integer values by table lookup. The macrocall is a predicate returning nonzero for true, zero for false. The isupper function is defined only where isascii8 is true and on the single non-ASCII value EOF. The function is nonzero if c is an uppercase letter. The uppercase letters are hexadecimal 41 through 5A, C0 through D6, and D8 through DE.

RELATED FUNCTIONS:

isalnum, isalpha, isascii, isascii8, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isxdigit.

isxdigit

Character classification (hexadecimal).

FORMAT:

```
# include <ctype.h>

int isxdigit (c)
int c;
```

ARGUMENTS:

c

Single-character value.

DESCRIPTION:

The isxdigit macrocall classifies ASCII-coded integer values by table lookup. The macrocall is a predicate returning nonzero for true, zero for false. The isxdigit function is defined only where isascii8 is true and on the single non-ASCII value EOF (see isascii8). The function is nonzero if c is a hexadecimal digit ([0 through 9], [A through F], or [a through f]).

RELATED FUNCTIONS:

isalnum, isalpha, isascii, isascii8, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper.

j0, j1, jn

Bessel functions.

FORMAT:

```
# include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn, (n, x);
double x;
int n;
```

ARGUMENTS:

x

Double-precision value.

n

Order of Bessel function.

DESCRIPTION:

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders. The jn function returns the Bessel function of x of the first kind of order n.

RELATED FUNCTIONS:

y0, y1, yn.

kill

Send a signal to a process or a group of processes.

FORMAT:

```
int kill (pid, sig)
int pid, sig;
```

ARGUMENTS:

pid

Process ID to be signaled.

sig

Signal to be sent.

DESCRIPTION:

The kill function sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by pid. The signal that is to be sent is specified by sig and is either one from the list given in the signal function, or 0. If sig is 0 (the null signal), error checking is performed but no signal is actually sent. This is useful to check the validity of pid. The concept of a super-user is not simulated.

The sending and receiving processes must belong to the same MOD 400 task group unless pid <= 0.

If pid is greater than zero, sig is sent only to the process whose ID is equal to pid.

If pid is 0 or -1, sig is sent to all processes of the sender's process group that are UNIX processes.

If pid is negative but not -1, sig is sent to all processes other than the sender whose MOD 400 task group ID is equal to the absolute value of pid and which are processes compatible with UNIX.

The kill subroutine considers a process group to be that set of tasks in a MOD 400 task group that have a trap handler connected and have the Intergroup Signal trap enabled. This is the trap used by the kill subroutine to send signals. All C programs enable this trap.

Because there is no super-user and the real and effective user IDs of a process are always the same; the definition of process group given above means that specifying -1 for the pid (process ID) formal parameter has the same results as specifying zero. That is, the signal is broadcast to the caller's process group.

DIAGNOSTICS:

The kill function fails and no signal is sent if:

- Sig is not a valid signal number [EINVAL].
- No process can be found corresponding to pid [ESRCH].

RETURN VALUE:

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the variables m4_errno and errno are set to indicate the error.

RELATED FUNCTIONS:

getpid, signal.

l3tol

Convert between three-byte integers and long integers.

FORMAT:

```
l3tol (lp, cp, n)
long *lp;
char *cp;
int n;
```

ARGUMENTS:

lp

Pointer to a list of long integers (output).

cp

Pointer to a list of three-byte integers (input).

n

Number of integers to be converted.

DESCRIPTION:

The l3tol function converts a list of n three-byte integers (packed into a character string pointed to by cp) into a list of long integers pointed to by lp.

RELATED FUNCTIONS:

ltol3.

l64a

Convert between long and base-64 ASCII.

FORMAT:

```
char *l64a(l)
long l;
```

ARGUMENTS:

l

Long value to be converted.

DESCRIPTION:

The l64a function is used to maintain numbers stored in base-64 ASCII. This is a notation by which long integers can be represented by up to six characters; each character represents a "digit" in a radix-64 notation.

The characters used to represent digits are . for 0, / for 1, 0 through 9 for 2-11, A through Z for 12-37, and a through z for 38-63.

The l64a function takes a long argument and returns a pointer to the corresponding base-64 representation.

NOTE

> The value returned by l64a is a pointer into a static buffer, the contents of which are overwritten by each call.

RELATED FUNCTIONS:

a64l.

ldexp

Exponential function.

FORMAT:

```
double ldexp (value, exp)
double value;
int exp;
```

ARGUMENTS:

value

Double-precision value.

exp

Exponent.

DESCRIPTION:

The ldexp function returns the quantity value*2^{exp}.

RELATED FUNCTIONS:

frexp, modf.

lgdiv

Divide long values.

FORMAT:

```
long lgdiv (a, b)
long a, b;
```

ARGUMENTS:

a

Long dividend.

b

Long divisor.

DESCRIPTION:

The lgdiv function performs division of the long value a by the long value b.

RELATED FUNCTIONS:

lgmul, lgrem, uldiv, ulrem.

lgmul

Multiply long values.

FORMAT:

```
long lgmul (a, b)
long a, b;
```

ARGUMENTS:

a

Long multiplier.

b

Long multiplicand.

DESCRIPTION:

The lgmul function performs multiplication of the long value a by the long value b.

RELATED FUNCTIONS:

lgdiv, lgrem, uldiv, ulrem.

<u>lgrem</u>

Remainder function.

FORMAT:

```
long lgrem (a, b)
long a, b;
```

ARGUMENTS:

a

Long dividend.

b

Long divisor.

DESCRIPTION:

The lgrem function returns the remainder of a/b.

RELATED FUNCTIONS:

lgdiv, lgmul, uldiv, ulrem.

link

Link to a file.

FORMAT:

```
int link (path1, path2)
char *path1, *path2;
```

ARGUMENTS:

$path_1$

Pathname of an existing file.

$path_2$

Pathname of the new directory entry to be created.

DESCRIPTION:

The link function creates a new link (directory entry) for an existing file.

The link function fails and no link is created if:

- A component of either path prefix is not a directory [ENOTDIR].
- A component of either path prefix does not exist [ENOENT].
- A component of either path prefix denies search access [EACCES].
- The file named by $path_1$ does not exist [ENOENT].
- The link named by $path_2$ exists [EEXIST].
- Pointer $path_2$ points to a null pathname [ENOENT].
- The requested link requires writing in a directory without write access [EACCES].
- The directory space limit has been reached [hex 0224].
- There is a media error [hex 01XX].
- There is not enough memory for buffers and structures [ENOMEM].

RETURN VALUE:

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the variable errno is set to indicate the error.

RELATED FUNCTIONS:

unlink.

localtime

Convert date and time to ASCII.

FORMAT:

```
# include <time.h>

struct tm *localtime (clock)
long *clock;
```

ARGUMENTS:

clock

Long integer pointer to the time in seconds since Jan. 1, 1970 (such as returned by time).

DESCRIPTION:

The localtime function returns a pointer to a structure containing the components of the time. The localtime function corrects for the time zone and possible daylight savings time.

The structure declaration from the include file is:

```
struct tm   {
       int     tm_sec;
       int     tm_min;
       int     tm_hour;
       int     tm_mday;
       int     tm_mon;
       int     tm_year;
       int     tm_wday;
       int     tm_yday;
       int     tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday - 0), year - 1900, day of year (0-365), and a flag that is nonzero if daylight saving time is in effect.

The external long variable timezone contains the difference, in seconds, between GMT and local standard time (in EST, timezone is 5*60*60); the external variable daylight is nonzero if and only if the standard U.S. daylight savings time conversion should be applied.

NOTE

The return values point to static data whose contents are overwritten by each call.

RELATED FUNCTIONS:

asctime, ctime, gmtime, time, tzset; see also the list_stz and set_stz commands.

log

Natural logarithm function.

FORMAT:

```
# include <math.h>

double log (x)
double x;
```

ARGUMENTS:

x

Double-precision value.

DESCRIPTION:

The log function returns the natural logarithm of x. X must be positive.

DIAGNOSTICS:

The log function returns a huge negative value and sets errno to EDOM when x is nonpositive.

RELATED FUNCTIONS:

exp, hypot, log10, pow, sinh, sqrt.

log10

Common logarithm function.

FORMAT:

```
# include <math.h>

double log10 (x)
double x;
```

ARGUMENTS:

x

Double-precision value.

DESCRIPTION:

The log10 function returns the common logarithm of x. X must be positive.

DIAGNOSTICS:

The log function returns a huge negative value and sets errno to EDOM when x is nonpositive.

RELATED FUNCTIONS:

exp, hypot, log, pow, sinh, sqrt.

longjmp

Non-local goto.

FORMAT:

```
# include <setjmp.h>

void longjmp (env, val)
jmp_buf env;
int val;
```

ARGUMENTS:

env

Pointer to the stack frame associated with the function that called setjmp and these registers:

- B7, B5, B4, B3, B2
- I
- R6, R5, R4, R3, R2, R1.

val

Value to be returned.

DESCRIPTION:

The longjmp function restores the environment saved by the most recent call to setjmp having env as its argument. It then returns in such a way that execution continues as if the call to setjmp had returned with the value val instead of zero (as is the case with the true return from setjmp). The function that called setjmp must not itself have returned in the interim. If longjmp is invoked with a val argument of zero, it behaves as if 1 had been used instead.

All accessible objects have values as of the time longjmp was called, except for objects of storage class register whose values have changed between the setjmp and longjmp calls. Variables allocated in registers retain the values they had when setjmp was called, while variables not allocated in registers retain the values they had when longjmp was called. Any program that relies on this treatment of register variables is implementation-dependent and, therefore, nonportable.

DIAGNOSTICS:

If the env argument does not contain a valid stack frame pointer, SIGSYS is signaled, without releasing any stack frames.

RELATED FUNCTIONS:

kill, setjmp, signal.

lsearch

Linear search and update.

FORMAT:

```
char *lsearch (key, base, nelp, width, compar)
char *key;
char *base;
int *nelp;
int width;
int (*compar)();
```

ARGUMENTS:

key

Pointer to the datum to be located in the table.

base

Pointer to the base of the table.

nelp

Address of an integer containing the number of items in the table. It is incremented if the item is added to the table.

width

Width of an element in characters.

compar

Name of the comparison routine.

DESCRIPTION:

The lsearch function is a linear search routine generalized from Knuth Algorithm Q. It returns a pointer into a table indicating the location at which a datum can be found. If the item does not occur, it is added at the end of the table.

The comparison routine is called with two character pointer arguments that point to the elements being compared. The routine must return zero if the items are equal and nonzero otherwise.

NOTE

Unspecified results can occur if there is not enough room in the table to add a new item.

RELATED FUNCTIONS:

bsearch, qsort.

ltol3

Convert between long integers and three-byte integers.

FORMAT:

```
ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

ARGUMENTS:

cp

Pointer to a list of three-byte integers (output).

lp

Pointer to a list of long integers (input).

n

Number of integers to be converted.

DESCRIPTION:

The ltol3 function converts a list of n long integers (pointed to by lp) into a list of three-byte integers (packed into a character string pointed to by cp).

RELATED FUNCTIONS:

l3tol.

malloc

Heap memory allocator.

FORMAT:

```
char *malloc (size)
unsigned int size;
```

ARGUMENTS:

size

Size of the desired memory block in characters.

DESCRIPTION:

The malloc function is part of a general-purpose heap memory allocation package. The malloc function returns a character pointer to the beginning of a double-word-aligned block of at least size characters. Such block are suitable for storing objects of any type.

The heap is managed by the C functions malloc, calloc, realloc, and free, and by the MOD 400 macrocalls Create Segment and Expand Segment. The heap is managed using a modified boundary-tag algorithm. This algorithm suffers little from memory fragmentation losses, yet is nearly as fast as a buddy-system algorithm.

The heap consists of one or more areas, each consisting of one or more segments. Heap areas are expanded, or new areas are created, as the need arises. Heap areas are never shrunk or deleted. However, when running in a fixed memory pool, the heap is restricted to a single, nonexpandable area whose size is specified by a Linker directive.

Memory is allocated in blocks of 16 bytes, plus 14 bytes for each allocated block. The block is not initialized.

DIAGNOSTICS:

If the heap does not contain enough memory, and cannot be sufficiently expanded, to meet the request, the variable errno is set to ENOMEM, the variable m4_errno is set to hexadecimal 1800+ENOMEM, and (char *) 0, a null character pointer, is returned.

RELATED FUNCTIONS:

calloc, free, realloc.

mcl

Execute MOD 400 system service macrocall.

FORMAT:

```
# include <XX_mcl.incl>

int mcl (function, context)
int function;
struct mcl_psb *context;
```

ARGUMENTS:

function

MOD 400 macrocall number.

context

Pointer to register context structure.

DESCRIPTION:

The mcl function performs the MOD 400 system service macrocall specified by the function argument. System service calls are defined in the MOD 400 System Programmer's Guide--Volume II.

The mcl function first loads the register context contained in the structure pointed to by the context argument. Then it executes the call with the function code given by the function argument. High-order bits of the context for base registers are ignored when loading the register context, so registers not used by the call need not be initialized.

The mcl function expects the address of the "fixed parameter block" for the Request Group macrocall to be made available in register B3's image (reg_b3) instead of in register B5's image (which does not exist), as is indicated in the System Programmer's Guide. Other macrocalls that require a parameter value in register B5 cannot be invoked via the mcl function.

The mcl function does not protect registers B6 and B7 from change. To invoke a macrocall that changes register B6, reset register B6 after return from mcl by calling an arbitrary (and possibly trivial) function, passing it no arguments that are not constants.

The XX_mcl.h include files are:

cl_mcl.h	Clock functions
dm_mcl.h	Data management functions
fm_mcl.h	File management functions
gp_mcl.h	Task group control functions
id_mcl.h	Identification and information functions
io_mcl.h	Input/output functions
mm_mcl.h	Memory management functions
mr_mcl.h	Message reporter functions
op_mcl.h	Operator interface functions
rq_mcl.h	Request and return functions
sf_mcl.h	System file functions
sm_mcl.h	Storage management functions
sw_mcl.h	External switch functions
sy_mcl.h	Semaphore functions
ts_mcl.h	Task control functions.

The include files also contain structure definitions for the argument and parameter structure blocks used by the various macrocalls. Each of the XX_mcl.h include files automatically includes the mcl.h include file if it has not already been included.

RETURN VALUE:

The mcl function returns the status code it received from the call. (Refer to the MOD 400 System Messages manual for a list of return status code values.) A value of zero always indicates successful completion of the call. Nonzero values usually indicate an error, but in some cases are informative only. Nonzero values are stored in the external variable m4_errno.

Upon return from the call, the mcl function saves the (possibly altered) register context in the structure pointed to by context.

NOTES

1. The XX_mcl.incl include file defines manifest constants for most macrocalls. The manifest constant for a macrocall named $XXX in the System Programmer's Guide--Volume II is named MCL$XXX in the include file. If the macrocall has a name longer than five characters (including the dollar sign), the name is shortened.

2. The include file also contains structure definitions for the argument and parameter structure blocks used by the various macrocalls. It includes these declarations of the parameter structure block used by the mcl function itself:

```
struct mcl_psb {
   int *reg_b4;
   int *reg_b3;
   int *reg_b2;
   int *reg_b1;
   int reg_r7;
   int reg_r6;
   int reg_r5;
   int reg_r4;
   int reg_r3;
   int reg_r2;
};
```

memccpy

Memory-to-memory copy.

FORMAT:

```
# include <memory.h>

unsigned char *memccpy (s1, s2, c, n)
unsigned char *s1, *s2;
unsigned char c;
int n;
```

ARGUMENTS:

s_1

Pointer to target memory area (output).

s_2

Pointer to source memory area (input).

c

Last character to copy (if found in s_2).

n

Number of characters to copy.

DESCRIPTION:

The memccpy function copies characters from memory area s_2 into s_1, stopping after the first occurrence of character c has been copied, or after n characters have been copied, whichever comes first. If n is less than or equal to zero, no characters are copied.

This function operates efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). This function does not check for the overflow of any receiving memory area.

RETURN VALUE:

This function returns a pointer to the character after the copy of c in s_1, or (unsigned char *) 0 if c was not found in the first n characters of s_2.

NOTE

This function is declared in the <memory.h> header file.

RELATED FUNCTIONS:

memchr, memcmp, memcpy, memset, umemchr, umemcmp, umemcpy, umemset.

memchr

Locate character in memory.

FORMAT:

```
# include <memory.h>

unsigned char *memchr (s, c, n)
unsigned char *s;
unsigned char c;
int n;
```

ARGUMENTS:

s

Pointer to memory area to check.

c

Character to seek.

n

Size of memory area in characters.

DESCRIPTION:

The memchr function returns a pointer to the first occurrence of character c within the first n characters of memory area s, or (unsigned char *) 0 if c does not occur.

This function operates efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character).

NOTE

This function is declared in the <memory.h> header file.

RELATED FUNCTIONS:

memccpy, memcmp, memcpy, memset, umemchr, umemcmp, umemcpy, umemset.

memcmp

Memory-to-memory compare.

FORMAT:

```
# include <memory.h>

int memcmp (s1, s2, n)
unsigned char *s1, *s2;
int n;
```

ARGUMENTS:

s_1

Pointer to first memory area to be compared.

s_2

Pointer to second memory area to be compared.

n

Size of memory areas in characters.

DESCRIPTION:

The memcmp function compares its arguments, looking at the first n characters only

This function operates efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). It executes without a stack frame of its own, and it makes use of commercial instructions.

RETURN VALUE:

This function returns an integer less than, equal to, or greater than zero, depending on whether s_1 is less than, equal to, or greater than s_2. If n is less than or equal to zero, equality is indicated.

NOTES

1. This function is declared in the <memory.h> header file.

2. The memcmp function uses 8-bit ASCII comparisons. Comparison proceeds from left to right until an unequal pair of characters is found or until all characters have been compared without finding an unequal pair. If an unequal pair is found, their ordering in the 8-bit ASCII code set determines the ordering of the two operands.

RELATED FUNCTIONS:

memccpy, memchr, memcpy, memset, umemchr, umemcmp, umemcpy, umemset.

memcpy

Memory-to-memory copy.

FORMAT:

```
# include <memory.h>

unsigned char *memcpy (s1, s2, n)
unsigned char *s1, *s2;
int n;
```

ARGUMENTS:

s_1

Pointer to target memory area (output).

s_2

Pointer to source memory area (input).

n

Number of characters to copy.

DESCRIPTION:

The memcpy function copies n characters from memory area s_2 to s_1.

This function operates efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). This function does not check for the overflow of any receiving memory area. It executes without a stack frame of its own.

RETURN VALUE:

This function returns s_1.

NOTES

1. This function is declared in the <memory.h> header file.

2. The memcpy function produces unspecified results if the memory areas overlap but are not identical.

memset

Initialize memory.

FORMAT:

```
# include <memory.h>

unsigned char *memset (s, c, n)
unsigned char *s;
unsigned char c;
int n;
```

ARGUMENTS:

s

Pointer to memory area to initialize.

c

Character to fill memory area.

n

Size of memory area in characters.

DESCRIPTION:

The memset function sets the first n characters in memory area s to the value of character c. If n is less than or equal to zero, no characters are set.

This function operates efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). This function does not check for the overflow of any receiving memory area. It executes without a stack frame of its own, and it makes use of commercial instructions.

RETURN VALUE:

This function returns *s.

NOTE

This function is declared in the <memory.h> header file.

mktemp

Make a unique file name.

FORMAT:

```
char *mktemp (template)
char *template;
```

ARGUMENTS:

template

Template character string plus six trailing Xs.

DESCRIPTION:

The mktemp function replaces template by a unique file name, and returns the address of the template. The template should look like a file name with six trailing Xs, which will be replaced with a letter and the current process ID. The letter is chosen so that the resulting name does not duplicate an existing file.

NOTE

It is possible to run out of letters.

RELATED FUNCTIONS:

getpid.

modf

Return fraction part of value.

FORMAT:

```
double modf (value, iptr)
double value, *iptr;
```

ARGUMENTS:

value

Double-precision value.

iptr

Pointer to integer part of value.

DESCRIPTION:

The modf function returns the signed fractional part of value and stores the integer part indirectly, through iptr.

RELATED FUNCTIONS:

frexp, ldexp.

open

Open for reading or writing.

FORMAT:

```
# include <stdio.h>

int open (path, oflag)
char *path;
int oflag;
```

ARGUMENTS:

path

Pathname of file to open.

oflag

Access flag (see below).

DESCRIPTION:

The open function opens a file descriptor for the named file and sets the file status flags according to the value of oflag. The path pointer refers to a pathname naming a file. Oflag values are constructed by performing a logical OR operation on flags from the following list:

O_RDONLY -- Open for reading only.

O_WRONLY -- Open for writing only.

O_RDWR -- Open for reading and writing.

O_CREAT -- Create a new file. If the file already exists, this flag has no effect.

O_EXCL -- Only meaningful in combination with O_CREAT; these flags together specify that the file must not already exist.

O_RDBIN -- Open binary file for reading only.

O_WRBIN -- Open binary file for writing only.

O_RDWRBIN -- Open binary file for reading and writing.

O_ABIN -- Same as O_WRBIN.

The file pointer (used to mark the current position within the file) is set to the beginning of the file.

This function also works with dynamic and device files. To open an interactive device file (such as a terminal), use the O_RDWR flag; to open a noninteractive device file (such as a printer), use O_RDONLY or O_WRONLY, as appropriate.

An I/O operation on a text file maps length-delimited records (MOD 400) to newline-delimited character streams (UNIX) and vice versa. An I/O operation on a binary file transfers a length-delimited record.

The new file descriptor remains open across exec calls.

No process can have more than 20 file descriptors open simultaneously.

The open function does not allocate a buffer until it is needed. When eventually needed, 136-character buffers are allocated for the user-in, user-out, and error-out files, and 512-character buffers for other files. The number of buffers ultimately allocated for a file is as follows:

- Binary files processed only by low-level I/O (read and write) get no buffers
- A user-in file processed only by low-level I/O gets no buffer
- String-relative files processed only by low-level I/O get no buffers
- All other files processed only by low-level I/O get one buffer each
- Files processed by high-level I/O get one more buffer than they would if processed only by low-level I/O.

An operation on a text file converts each record to a character stream ending with a newline character, and vice versa. An operation on a binary file transfers fixed-length records directly. In either case, the file is treated as a stream of characters processed by the getc and putc macrocalls. (The buffering required precludes using both getc and putc on a file opened for updating.)

RETURN VALUE:

Upon successful completion, a file descriptor (a nonnegative integer) is returned. Otherwise, a value of -1 is returned and the variables errno and m4_errno are set to indicate the error returned from MOD 400.

RELATED FUNCTIONS:

close, creat, dup, fcntl, read, write.

pause

Suspend process until signal.

FORMAT:

```
pause ()
```

ARGUMENTS:

None.

DESCRIPTION:

The pause function suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

The default response to the receipt of a signal is the termination of the receiving process. The call to signal specifies alternatives:

- Ignore a signal
- Call a function with the signal number as the argument
- Designate or reinstate the default (termination).

If the signal causes termination of the calling process, pause will not return.

If the signal is caught and control is returned from the signal-catching function (see signal), the calling process resumes execution from the point of suspension (the call to pause), with a return value of -1 from pause, the value of errno set to EINTR, and the value of m4_errno set appropriately.

RELATED FUNCTIONS:

alarm, kill, signal, wait.

perror

Print system error message.

FORMAT:

```
void perror (s[, a[, b[, c]]])

char *s[, *a[, *b[, *c]]];

extern int errno;
```

ARGUMENTS:

s

Name of the program that incurred the error.

a, b, c

Optional parameters to specialize the text of the message. Refer to the MOD 400 System Messages manual.

errno

Error number.

DESCRIPTION:

The perror function produces a short error message on the error-out file, describing the last error encountered during a function. Text appears as follows:

1. The argument string s
2. A colon
3. A blank
4. The message text
5. A newline character.

The argument string should be the name of the program that incurred the error. The error number is taken from the external variable errno, which is set when errors occur but not cleared when nonerroneous calls are made.

The text of the message is obtained from the MOD 400 error message libraries. If errno has a value in the range 1 to 255, hexadecimal 1800 is added to it to obtain the MOD 400 error number. If errno is not in this range, it is used as is. If the MOD 400 error message libraries do not contain a message for that error code, the text "Error number X.", where X is the value of errno, is used instead.

RELATED FUNCTIONS:

errno, sys_errlist, sys_nerr.

pipe

Intergroup channel.

FORMAT:

```
int pipe (fildes)
int fildes[2];
```

ARGUMENTS:

fildes

File descriptor.

DESCRIPTION:

The pipe function creates a pipe and returns two file descriptors, fildes[0] (for reading) and fildes[1] (for writing).

Write operations are buffered up to 5120 characters and blocked. A read operation on fildes[0] receives data written to fildes[1] on a first-in, first-out basis.

No group can have more than 20 file descriptors open simultaneously.

RETURN VALUE:

Upon successful completion, the pipe function returns a value of 0. Otherwise, a value of -1 is returned, and the variable errno is set to indicate the error.

DIAGNOSTICS:

The pipe function fails if it is called and 19 or more file descriptors are currently open [EMFILE].

RELATED FUNCTIONS:

read, write.

posr

Position record pointer.

FORMAT:

```
# include <ufas.h>

int posr(cmd,fildes,rtype,keyptr,keytype)
int cmd, fildes, rtype, keytype;
record_key *keyptr;
```

ARGUMENTS:

cmd

Command (see "Description").

fildes

Open file descriptor obtained from a creat, open, dup, or fcntl function.

rtype

Record type: 0 for read-pointer operations, or -1 for write-pointer operations.

keyptr

Pointer to key value.

keytype

Optional key type:

PRIMARY -- For indexed files; keyvalue type is (char *)

RELATIVE -- For relative files; keyvalue type is (long *)

CALC -- For random files; keyvalue type is (char *)

SIMPLE -- For sequential or dynamic files; keyvalue type is (long *)

ALT -- For alternate index; keyvalue type is (char *)

CURRENT -- Current key of usage.

DESCRIPTION:

The posr function positions a read or write pointer within an open file, according to a key value.

Acceptable values for cmd are:

```
RD_EQ  -- Position read pointer equal to
RD_GR  -- Position read pointer greater than
RD_GE  -- Position read pointer greater than or equal to
RD_FWD -- Position read pointer forward
RD_BWD -- Position read pointer backward
WR_EQ  -- Position write pointer equal to
WR_GR  -- Position write pointer greater than
WR_GE  -- Position write pointer greater than or equal to
WR_FWD -- Position write pointer forward
WR_BWD -- Position write pointer backward
```

The record_key data type is defined in the ufas.h file as:

```
typedef union {
             unsigned long n;
             unsigned char s[];
              } record_key;
```

The member n (32 bits) is used for simple or relative keys. The member s (variable) is used for other keys.

A simple key is constructed from the control-interval number and line number of a record according to this formula:

```
key = (256 * CI) + (line)
```

For a relative key, n is the value directly. For a primary or CALC key, s is the key value directly. Always specify an alternate key where appropriate.

RETURN VALUE:

Upon successful completion, a value of 0 is returned. Otherwise, the variables errno and m4_errno are set to indicate the error, and a value of -1 is returned.

RELATED FUNCTIONS:

getr, putr, ucreat.

pow

Power function.

FORMAT:

```
# include <math.h>

double pow (x, y)
double x, y;
```

ARGUMENTS:

x, y

Double-precision values.

DESCRIPTION:

The pow function returns x^y. The values of x and y cannot both be zero. If x is less than or equal to zero, y must be an integer.

DIAGNOSTICS:

The pow function returns a huge value when the correct value would overflow. A truly outrageous argument can also result in errno being set to ERANGE.

The pow function returns a huge negative value and sets errno to EDOM when x is nonpositive and y is not an integer, or when x and y are both zero.

RELATED FUNCTIONS:

exp, hypot, log, sinh, sqrt.

printf

Format output.

FORMAT:

```
# include <stdio.h>

int printf (format [, arg] ... )
char *format;
```

ARGUMENTS:

format

Format string.

arg

Optional argument to be printed.

DESCRIPTION:

The printf function writes output to the user-out file. It is equivalent to a call to fprintf with the argument stdout inserted before the arguments to fprintf.

For more information on this function, refer to the description of fprint.

RELATED FUNCTIONS:

ecvt, fprintf, putc, scanf, sprintf.

pthto6

Convert UNIX pathname to MOD 400.

FORMAT:

```
int pthto6 (inpath, outpath)
char *inpath, *outpath;
```

ARGUMENTS:

inpath

Pointer to a null-terminated character string. The string can be a UNIX pathname, a MOD 400 pathname, or a combination of both.

outpath

Pointer to a string at least 60 characters long.

DESCRIPTION:

The pthto6 function maps pathnames compatible with UNIX to pathnames compatible with MOD 400. It detects invalid characters, invalid directory names, and overlong pathnames.

RETURN VALUE:

If no error is encountered, a space and null character are appended to the output pathname. The return value is the length of the output pathname, including the space and null terminators. (Therefore, pthto6, when successful, always returns a value greater than zero.)

DIAGNOSTICS:

The pthto6 function terminates when it encounters an error, a space, or a null character. The input and output strings can be the same.

If pthto6 finds an error, it returns a value of -1. A space and null character are appended to the output path up to that point, and the variable errno is set to ENOENT. The variable m4_errno is set to 0201 ("The pathname violates naming conventions").

putc

Put a character on a file.

FORMAT:

```
# include <stdio.h>

int putc (c, file)
char c;
FILE *file;
```

ARGUMENTS:

c

Character to be appended to the file.

file

File pathname.

DESCRIPTION:

The putc function appends the character c to the buffer associated with the named output file, writing the buffer whenever it is full.

RETURN VALUE:

The putc function returns the character appended.

DIAGNOSTICS:

This function returns the constant EOF when it encounters an error. Since this is a good integer, ferror should be used to detect putw errors.

NOTE

Because it is a macrocall, putc treats incorrectly a file argument with side effects, for example, putc(c, *f++); .

RELATED FUNCTIONS:

ferror, fopen, fputc, fwrite, getc, printf, putchar, puts, putw.

putchar

Put character on stdout file.

FORMAT:

```
# include <stdio.h>

putchar (c)
```

ARGUMENTS:

c

Character to be appended to the file.

DESCRIPTION:

The putchar(c) function is defined as putc(c, stdout).

DIAGNOSTICS:

This function returns the constant EOF when it encounters an error. Since this is a good integer, ferror should be used to detect putw errors.

RELATED FUNCTIONS:

ferror, fopen, fputc, fwrite, getc, printf, putc, puts, putw.

putr

Put record.

FORMAT:

```
# include <ufas.h>

int putr(cmd,fildes,rptr,rlen,rtype[,keyptr,keytype])
int cmd, fildes, rlen, rtype[, keytype];
char *rptr;
record_key *keyptr;
```

ARGUMENTS:

cmd

Command (see "Description").

fildes

Open file descriptor obtained from a creat, open, dup, or fcntl function.

rptr

Pointer to a record area from which the record to be written is obtained.

rlen

Number of characters to write.

rtype

Output record type: a decimal value from 0 through 3999; set to 0 if no specific record type is desired.

keyptr

Pointer to key value.

keytype

Optional key type:

PRIMARY -- For indexed files; keyvalue type is (char *)

RELATIVE -- For relative files; keyvalue type is (long *)

CALC -- For random files; keyvalue type is (char *)

SIMPLE -- For sequential or dynamic files; keyvalue type is (long *)

ALT -- For alternate index; keyvalue type is (char *)

CURRENT -- For current key of usage.

DESCRIPTION:

The putr function writes or updates a record in a file, according to a key value.

Acceptable values for cmd are:

WR_NXT -- Write next record
WR_KEY -- Write with key
RW_CURR -- Rewrite current record
RW_KEY -- Rewrite record with key.

The last two arguments (keyptr and keytype) are optional for sequential write operations.

If you omit the keytype argument, the key type is determined by the file organization:

File organization	Key type
Sequential	SIMPLE
Relative (all types)	RELATIVE
Random	RANDOM
Dynamic	SIMPLE
Indexed	PRIMARY

The record_key data type is defined in the ufas.h file as:

```
typedef union {
            unsigned long n;
            unsigned char s[];
             } record_key;
```

The member n (32 bits) is used for simple or relative keys. The member s (variable) is used for other keys.

A simple key is constructed from the control-interval number and line number of a record according to this formula:

key = (256 * CI) + (line)

For a relative key, n is the value directly. For a primary or CALC key, s is the key value directly. Always specify an alternate key where appropriate.

When dealing with indexed files, specify record types that are a subset of record types specified when the files were created. You cannot change a record type by rewriting it.

Example:

The following sample fragment of code modifies the file MYREL.

```
# include <stdio.h>
# include <ufas.h>
main()
{
   long key;                          /* Relative key */
   int fildes;
   register k;
   fildes=open("MYREL",O_WRONLY);
   for (k=0; k<5; k++)   /* Write 5 successive records */
     putr(WR_NXT,fildes,"aaa",3,0);
/* Write records 7 and 9 */
   key = 7;
   putr(WR_KEY,fildes,"bbb",3,0,&key); /* Relative key */
   key = 9;              /* is default, or you can specify*/
   putr(WR_KEY,fildes,"ccc",3,0,&key,RELATIVE);   /* it*/
/* Reposition pointer to record 8; write next record (8)*/
   key = 7;
   posr(WR_GR,fildes,0,&key);
   putr(WR_NXT,fildes,"qqq",3,0);
/* Move pointer 3 records forward from current position */
   key = 3;
   posr(WR_FWD,fildes,0,&key);
   putr(WR_NXT,fildes,"qqq",3,0);
/* Now use simple key (you must specify) to write a record */
   key = 0x020F;
   putr(WR_KEY,fildes,"kkk",3,0,&key,SIMPLE);
   close(fildes);
}
```

After this code executes, the file MYREL contains:

```
aaa
aaa
aaa
aaa
aaa
bbb
qqq
ccc
qqq
kkk
```

RETURN VALUE:

Upon successful completion, a value of 0 is returned. Otherwise, the variables errno and m4_errno are set to indicate the error, and a value of -1 is returned.

RELATED FUNCTIONS:

getr, posr, ucreat.

puts

Put string on stdout file.

FORMAT:

```
# include <stdio.h>

int puts (s)
char *s;
```

ARGUMENTS:

s

String to be written to the file.

DESCRIPTION:

The puts function copies the null-terminated string s to the user-out file and appends a newline character.

This function does not copy the terminating null character.

DIAGNOSTICS:

This function returns EOF on error.

NOTE

The puts function appends a newline character.

RELATED FUNCTIONS:

ferror, fflush, fopen, fputs, fwrite, gets, printf, putc.

putw

Put a word on a file.

FORMAT:

```
# include <stdio.h>

putw (w, file)
int w;
FILE *file;
```

ARGUMENTS:

w

Integer to be written to the file.

file

File pathname.

DESCRIPTION:

The putw function appends the integer w to the output file. The putw function neither assumes nor causes special alignment in the file.

DIAGNOSTICS:

This function returns the constant EOF when it encounters an error. Since this is a good integer, ferror should be used to detect putw errors.

RELATED FUNCTIONS:

ferror, fopen, fputc, fwrite, getc, printf, putc, putchar, puts.

qsort

Quicker sort.

FORMAT:

```
qsort (base, nelem, width, compar)
char *base;
unsigned nelem;
int width;
int (*compar)( );
```

ARGUMENTS:

base

Pointer to the base of the data.

nelem

Number of elements.

width

Width of each element in characters.

compar

Name of the comparison routine.

DESCRIPTION:

The qsort function is an implementation of the quicker-sort algorithm. The comparison routine is called with two character pointer arguments, which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 depending on whether the first argument is less than, equal to, or greater than the second.

RELATED FUNCTIONS:

sort, bsearch, lsearch, strcmp.

rand

Generate random numbers.

FORMAT:

```
int rand()
```

ARGUMENTS:

None.

DESCRIPTION:

The rand function uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudorandom numbers in the range from 0 to 2^{15} -1.

RELATED FUNCTIONS:

srand.

read

Read from a file.

FORMAT:

```
int read (fildes, buf, nchar)
int fildes;
char *buf;
unsigned nchar;
```

ARGUMENTS:

fildes

File descriptor obtained from a creat, open, dup, fcntl, or pipe function call.

buf

Pointer to buffer.

nchar

Number of characters to read.

DESCRIPTION:

The read function attempts to read nchar characters from the file associated with fildes into the buffer pointed to by buf.

The read function recognizes EOT (Control-D) as an end-of-file character when received at the beginning of a line read from an interactive device. This is consistent with UNIX practice.

Text file end-of-file processing is compatible with UNIX. End-of-file conditions for binary files are the same as for text files.

The read function does not allocate a buffer until it is needed. The function allocates 136-character buffers for the user-in, user-out, and error-out files and 512-character buffers for other files. The number of buffers ultimately allocated for a file is as follows:

- Binary files processed only by low-level I/O (read and write) get no buffers

- A user-in file processed only by low-level I/O gets no buffer
- String-relative files processed only by low-level I/O get no buffers
- All other files processed only by low-level I/O get one buffer each
- Files processed by high-level I/O get one more buffer than they would if processed only by low-level I/O.

RETURN VALUE:

Upon successful completion, a nonnegative integer is returned indicating the number of characters actually read and placed in the buffer. A value of 0 is returned when an end of file has been reached. Otherwise, a -1 is returned and the variables errno and m4_errno are set to indicate the error.

RELATED FUNCTIONS:

creat, dup, fcntl, open, pipe.

realloc

Reallocate heap memory.

FORMAT:

```
char *realloc (ptr, size)
char *ptr;
assigned size;
```

ARGUMENTS:

ptr

Pointer to memory area to be reallocated.

size

New size, in characters.

DESCRIPTION:

The realloc function changes the size of the block pointed to by ptr to size characters and returns a pointer to the (possibly moved) block. The contents are unchanged up to the lesser of the new and old sizes.

The realloc function returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS:

If the heap does not contain enough memory, and cannot be sufficiently expanded to meet the request, the variable errno is set to ENOMEM, the variable m4_errno is set to hexadecimal 1800+ENOMEM, and (char *) 0, a null character pointer, is returned. When realloc returns a null pointer, the block pointed to by ptr may have been destroyed.

RELATED FUNCTIONS:

calloc, free, malloc.

runl

Create a new process.

FORMAT:

```
int runl(path,arg0,arg1,...,argn,(unsigned char *) 0)
unsigned char *path, *arg0, *arg1, ..., *argn;
```

ARGUMENTS:

path

Pointer to a pathname that identifies the new process bound unit.

arg_0, arg_1, ..., arg_n

Pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, at least arg_0 must be present and point to a string that is the same as path (or its file-name component).

DESCRIPTION:

The runl function creates a new process. The new process is constructed from an ordinary bound unit called the new process bound unit.

When a C program is executed, it is called as follows:

```
int main (argc, argv, envp)
int argc;
unsigned char **argv, **envp;
```

where argc is the argument count and argv is an array of character pointers to the arguments themselves. By convention, argc is at least one and argv[0] points to a string containing the name of the file.

When a run function or the MOD 400 command processor creates a process, a pointer to the environment of the calling process is placed in the global cell:

```
extern unsigned char **environ;
```

It is used to pass the environment of the calling process to the new process.

The environment provided is the C environment lines of the MOD 400 task group with the environment lines for HOME and PATH appended. (If the task group's C environment already contains an environment line for HOME or PATH, that environment line will take precedence.) The default PATH environment line specifies the referencing directory, the working directory, >SYSLIB1, and >SYSLIB2, in that order. The referencing directory is the directory from which the main program itself was loaded.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set. For those file descriptors that remain open, the file currency (read or write) is unchanged.

The new process inherits nothing else from the calling process.

The runl function fails and returns to the calling process if:

- One or more components of the pathname do not exist [ENOENT].
- A directory-name component of path is not a directory [ENOTDIR].
- List access is denied for a directory named in path [EACCES].
- The new process bound unit is not a bound unit, or the calling process lacks execute access to it [EACCES].
- The new process requires more memory than is allowed [ENOMEM].
- The number of characters in the argument list for the new process is greater than 5120 characters [E2BIG].
- The path, argv, or envp argument points to an invalid address [EFAULT].

RETURN VALUE:

Upon successful completion, runl returns the process ID of the new process to the calling process. Otherwise, the return value is -1, and the variables m4_errno and errno are set to indicate the error.

RELATED FUNCTIONS:

runlp, runv, runvp, exit, fork, getenv; see also the dl_env, get_env, list_env, and set_env commands.

runlp

Create a new process.

FORMAT:

```
int runlp(file,arg0,arg1,...,argn(unsigned char *)0)
unsigned char *file, *arg0, *arg1, ..., *argn;
```

ARGUMENTS:

file

Pointer to the filename of the new process bound unit.

arg_0, arg_1, ..., arg_n

Pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least arg_0 must be present and point to a string that is the same as path (or its filename component).

DESCRIPTION:

The runlp function creates a new process. The new process is constructed from an ordinary bound unit called the new process bound unit.

When a C program is executed, it is called as follows:

```
int main (argc, argv, envp)
int argc;
unsigned char **argv, **envp;
```

where argc is the argument count and argv is an array of character pointers to the arguments themselves. By convention, argc is at least one and argv[0] points to a string containing the name of the file.

When a run function or the MOD 400 command processor creates a process, a pointer to the environment of the calling process is placed in the global cell:

```
extern unsigned char **environ;
```

It is used to pass the environment of the calling process to the new process.

The environment provided is the C environment lines of the MOD 400 task group with the environment lines for HOME and PATH appended. (If the task group's C environment already contains an environment line for HOME or PATH, that environment line will take precedence.) The default PATH environment line specifies the referencing directory, the working directory, >SYSLIB1, and >SYSLIB2, in that order. The referencing directory is the directory from which the main program itself was loaded.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set. For those file descriptors that remain open, the file currency (read or write) is unchanged.

The new process inherits nothing else from the calling process.

The runlp function fails and returns to the calling process if:

- One or more components of the pathname do not exist [ENOENT].
- A directory-name component of path is not a directory [ENOTDIR].
- List access is denied for a directory named in path [EACCES].
- The new process bound unit is not a bound unit, or the calling process lacks execute access to it [EACCES].
- The new process requires more memory than is allowed [ENOMEM].
- The number of characters in the argument list for the new process is greater than 5120 characters [E2BIG].
- The path, argv, or envp argument points to an invalid address [EFAULT].

RETURN VALUE:

Upon successful completion, runlp returns the process ID of the new process to the calling process. Otherwise, the return value is -1, and the variables m4_errno and errno are set to indicate the error.

RELATED FUNCTIONS:

runl, runv, runvp, exit, fork, getenv; see also the dl_env, get_env, list_env, and set_env commands.

runv

Execute a bound unit.

FORMAT:

```
int runv (path, argv)
unsigned char *path, *argv [];
```

ARGUMENTS:

path

Pointer to a pathname that identifies the new process bound unit.

argv

Array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, argv must have at least one member, and it must point to a string that is the same as path (or its file name component). The array is terminated by a null character pointer.

DESCRIPTION:

The runv function creates a new process. The new process is constructed from an ordinary bound unit called the new process bound unit.

When a C program is executed, it is called as follows:

```
int main (argc, argv, envp)
int argc;
unsigned char **argv, **envp;
```

where argc is the argument count and argv is an array of character pointers to the arguments themselves. By convention, argc is at least one and argv[0] points to a string containing the name of the file.

When a run function or the MOD 400 command processor creates a process, a pointer to the environment of the calling process is placed in the global cell:

```
extern unsigned char **environ;
```

It is used to pass the environment of the calling process to the new process.

The environment provided is the C environment lines of the MOD 400 task group with the environment lines for HOME and PATH appended. (If the task group's C environment already contains an environment line for HOME or PATH, that environment line will take precedence.) The default PATH environment line specifies the referencing directory, the working directory, >SYSLIB1, and >SYSLIB2, in that order. The referencing directory is the directory from which the main program itself was loaded.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set. For those file descriptors that remain open, the file currency (read or write) is unchanged.

The new process inherits nothing else from the calling process.

The runv function fails and returns to the calling process if:

- One or more components of the pathname do not exist [ENOENT].
- A directory-name component of path is not a directory [ENOTDIR].
- List access is denied for a directory named in path [EACCES].
- The new process bound unit is not a bound unit, or the calling process lacks execute access to it [EACCES].
- The new process requires more memory than is allowed [ENOMEM].
- The number of characters in the argument list for the new process is greater than 5120 characters [E2BIG].
- The path, argv, or envp argument points to an invalid address [EFAULT].

RETURN VALUE:

Upon successful completion, runv returns the process ID of the new process to the calling process. Otherwise, the return value is -1, and the variables m4_errno and errno are set to indicate the error.

RELATED FUNCTIONS:

runl, runlp, runvp, exit, fork, getenv; see also the dl_env, get_env, list_env, and set_env commands.

runvp

Create a new process.

FORMAT:

```
int runvp (file, argv)
unsigned char *file, *argv []
```

ARGUMENTS:

file

Pointer to the filename of the new process bound unit.

argv

Array of character pointers to null-terminated strings. These strings constitute the argument list available to the new task. By convention, argv must have at least one member, and it must point to a string that is the same as path (or its file name component). The array is terminated by a null character pointer.

DESCRIPTION:

The runvp function creates a new process. The new process is constructed from an ordinary bound unit called the new process bound unit.

When a C program is executed, it is called as follows:

```
int main (argc, argv, envp)
int argc;
unsigned char **argv, **envp;
```

where argc is the argument count and argv is an array of character pointers to the arguments themselves. By convention, argc is at least one and argv[0] points to a string containing the name of the file.

When a run function or the MOD 400 command processor creates a process, a pointer to the environment of the calling process is placed in the global cell:

```
extern unsigned char **environ;
```

It is used to pass the environment of the calling process to the new process.

The environment provided is the C environment lines of the MOD 400 task group with the environment lines for HOME and PATH appended. (If the task group's C environment already contains an environment line for HOME or PATH, that environment line will take precedence.) The default PATH environment line specifies the referencing directory, the working directory, >SYSLIB1, and >SYSLIB2, in that order. The referencing directory is the directory from which the main program itself was loaded.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set. For those file descriptors that remain open, the file currency (read or write) is unchanged.

The new process inherits nothing else from the calling process.

The runvp function fails and returns to the calling process if:

- One or more components of the pathname do not exist [ENOENT].
- A directory-name component of path is not a directory [ENOTDIR].
- List access is denied for a directory named in path [EACCES].
- The new process bound unit is not a bound unit, or the calling process lacks execute access to it [EACCES].
- The new process requires more memory than is allowed [ENOMEM].
- The number of characters in the argument list for the new process is greater than 5120 characters [E2BIG].
- The path, argv, or envp argument points to an invalid address [EFAULT].

RETURN VALUE:

Upon successful completion, runvp returns the process ID of the new process to the calling process. Otherwise, the return value is -1, and the variables m4_errno and errno are set to indicate the error.

RELATED FUNCTIONS:

runl, runlp, runv, exit, fork, getenv; see also the dl_env, get_env, list_env, and set_env commands.

same_file

Determine if two pathnames designate the same file.

FORMAT:

```
int same_file (path1, path2)
unsigned char *path1, *path2;
```

ARGUMENTS:

$path_1$

First (null-terminated) pathname to be checked.

$path_2$

Second (null-terminated) pathname to be checked.

DESCRIPTION:

The same_file function determines whether two strings naming files name the same file or different files.

RETURN VALUE:

If $path_1$ names a file that exists and $path_2$ also names that file, the value 1 is returned. If $path_1$ and $path_2$ both name files that exist but are not the same file, the value zero is returned. If $path_1$ does not name an existing file, m4_errno is set appropriately and the value -1 is returned. If $path_1$ names an existing file but $path_2$ does not, m4_errno is set appropriately and the value -2 is returned.

sbrk

Change data segment space allocation.

FORMAT:

```
char *sbrk (incr)
int incr;
```

ARGUMENTS:

incr

Number of characters to add to brk value.

DESCRIPTION:

The sbrk function is used to dynamically change the amount of space allocated for the calling process's break segment (see exec). The change is made by resetting the process's break value. The break value is the address of the first location beyond the end of the break segment. The amount of allocated space increases as the break value increases.

The sbrk function adds incr characters to the break value and changes the allocated space accordingly. The argument incr can be negative, in which case the amount of allocated space is decreased.

RETURN VALUE:

Upon successful completion, sbrk returns the old break value. Otherwise, a value of (char *) -1 is returned and the variables errno and m4_errno are set to indicate the error.

DIAGNOSTICS:

The sbrk function fails without making any change in the allocated space if such a change would result in more space being allocated than is allowed by MOD 400 [ENOMEM].

NOTES

1. The first call to sbrk creates a break segment. It may be a giant segment (larger than 128K characters). If this segment cannot be created for any reason, errno is set to ENOMEM and (char *) -1 is returned. MOD 400 chooses where to place it.

2. When a C task runs outside of a swappool, MOD 400 allocates memory from the task's memory pool instead of creating a segment; subsequent calls cannot increase the size of the break segment.

RELATED FUNCTIONS:

brk, exec family.

scanf

Formatted input conversion.

FORMAT:

```
# include <stdio.h>

scanf (format [,pointer]...)
char *format;
```

ARGUMENTS:

format

Control string format.

pointer

Set of arguments indicating where the converted input should be stored.

DESCRIPTION:

The scanf function reads from the standard input file stdin. This function reads characters, interprets them according to a format, and stores the results in its arguments. It requires a control string format and a set of optional pointer arguments indicating where the converted input should be stored.

The scanf function is equivalent to a call to fscanf with the argument stdout inserted before the arguments to scanf.

For more information on this function, refer to the description of the fscanf function.

RELATED FUNCTIONS:

atof, fscanf, getc, printf, sscanf.

send_sig

Send a signal to a process.

FORMAT:

```
int send_sig, (group, task, sig)
int group, *task, sig;
```

ARGUMENTS:

group

Task group ID of process to receive signal. A value of -1 means the caller's own group.

task

Address of task control block of process to receive signal.

sig

Signal number to be sent.

DESCRIPTION:

The send_sig function sends a signal to a process. The signal to be sent is either one from the list given in the description of the signal function or zero. If sig is zero, error checking is performed but no signal is actually delivered to the specified process. This can be used to determine if the specified process exists.

RETURN VALUE:

This function returns zero if the signal is successfully delivered to the specified process. Otherwise, the external variables errno and m4_errno are set to indicate the cause of the error, and -1 is returned.

DIAGNOSTICS:

The send_sig function fails if:

- The sig argument is an invalid signal number [EINVAL].
- The specified process does not exist [ESRCH].

RELATED FUNCTIONS:

exec family, fork, getpgrp, getptcb, gettcb, run family.

setbuf

Assign buffering to a file.

FORMAT:

```
# include <stdio.h>

setbuf (file, buf)
FILE *file;
char *buf;
```

ARGUMENTS:

file

File pathname.

buf

Pointer to buffer address.

DESCRIPTION:

The setbuf function is used after a file has been opened but before it is read or written. It causes the character array buf to be used instead of an automatically allocated buffer.

A manifest constant BUFSIZ tells how big an array is needed:

```
char buf[BUFSIZ];
```

RELATED FUNCTIONS:

fopen, getc, putc.

setgrent

Set group record entry.

FORMAT:

```
# include <grp.h>

void setgrent ()
```

ARGUMENTS:

None.

DESCRIPTION:

A call to setgrent has the effect of making the next call to getgrent a "first" call.

RELATED FUNCTIONS:

endgrent, getgrent, getgrgid, getgrnam, getlogin, getpwent, group.

setjmp

Non-local goto.

FORMAT:

```
# include <setjmp.h>

int setjmp (env)
jmp_buf env;
```

ARGUMENTS:

env

Pointer to the stack frame and these registers:

- B7, B5, B4, B3, B2
- I
- R6, R5, R4, R3, R2, R1.

DESCRIPTION:

The setjmp function saves its stack environment in env for later use by longjmp.

This routine is useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

RETURN VALUE:

This function returns the value zero.

RELATED FUNCTIONS:

kill, longjmp, signal.

setkey

DES encryption.

FORMAT:

```
setkey (key)
char *key;
```

ARGUMENTS:

key

Sixty-four-character binary array.

DESCRIPTION:

The setkey function is based on the National Bureau of Standards Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search. The setkey and encrypt function provides access to the actual DES algorithm. The key argument is a 64-character binary array. If this string is divided into groups of eight, the low-order bit in each group is ignored, leading to a 56-bit key which is set into the machine.

RELATED FUNCTIONS:

crypt, encrypt.

NOTE

The return value points to static data that is overwritten by each call.

setprint

Set the print attribute of a stream.

FORMAT:

```
# include <stdio.h>

void setprint (stream)
FILE *stream;
```

ARGUMENTS:

stream

Name of the file.

DESCRIPTION:

The setprint function is used after a stream is opened but before it is written. It turns on the print attribute of a stream. This attribute is meaningful only for text mode output files (that is, opened without the O_BINARY open flag or the "b" fopen type). It causes the stream to be written with MOD 400 print control characters.

The print attribute is implicitly set for the stdin, stdout, and stderror files open at the time main is called. It is also implicitly set for serial and line printer device files. String-relative files are always in binary mode.

RELATED FUNCTIONS:

fopen, open, write.

setpwent

Rewind password file.

FORMAT:

```
# include <pwd.h>

void setpwent ()
```

ARGUMENTS:

None.

DESCRIPTION:

The setpwent function resets the password file. A call to setpwent has the effect of making the next call to getpwent a "first" call.

The effect is to cause the caller to perceive the system as a single-user UNIX system.

RELATED FUNCTIONS:

endpwent, getpwent, getpwnam, getpwuid.

signal

Specify what to do upon receipt of a signal.

FORMAT:

```
# include <signal.h>

int (*signal (sig, func))()
int sig;
int (*func)();
```

ARGUMENTS:

sig

Signal to be processed.

func

SIG_DFL, SIG_IGN, or a function address (see below).

DESCRIPTION:

The signal function allows the calling process to choose one of three ways to handle the receipt of a specific signal. The sig argument specifies the signal and the func argument specifies the choice.

A signal is generated by some abnormal event, such as a Megabus error, receipt of a kill, or your pressing Break. Normally, all signals terminate the process. The signal function allows a process to ignore a signal or cause an interrupt to a specified location.

The sig argument can be assigned from the following:

SIGHUP	01	Hangup
SIGINT	02	Interrupt
SIGQUIT	03*	Quit
SIGILL	04*	Invalid instruction
SIGTRAP	05*	Trace trap (not reset when caught)
SIGIOT	06*	IOT instruction
SIGEMT	07*	EMT instruction
SEGFPE	08*	Floating-point exception
SIGKILL	09	Kill (cannot be caught or ignored)
SIGBUS	10*	Megabus error
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Invalid argument to function
SIGALRM	14	Alarm clock

SIGTERM	15	Software termination signal
SIGUSR1	16	User-defined signal 1
SIGUSR2	17	User-defined signal 2
SIGCLD	18	Death of a child (see note)
SIGPWR	19	Power failure recovery (not reset when caught)

The signals with an asterisk cause a memory dump to the file CORE (generated by the MOD 400 Dump Edit utility invoked with the argument -ME) in the working directory of the receiving process unless caught or ignored.

NOTE

There are two signals that behave differently:

SIGCLD	18	death of a child process
SIGPWR	19	power failure recovery

SIGCLD (18) is always reset when caught. SIGPWR (19) is not reset when caught. Their use in new programs is strongly discouraged.

For both signals, SIG_DFL is treated as SIG_IGN.

A parent process may use the signal function to ignore the receipt of the SIGCLD signal. When a child process terminates, this change of state is used to initiate actions such as the handling of the wait of the parent process. Sending the SIGCLD signal is neither needed nor used in the child process termination actions.

The actions prescribed by the sig argument are:

- SIG_DFL -- Set the default that terminates process upon receipt of signal. Upon receipt of the signal sig, the receiving process is to be terminated with the following consequences:

  - All of the receiving process's open file descriptors are closed.

  - If the parent process of the receiving process is executing a wait, it is notified of the termination of the receiving process and the signal's number is made available to the parent.

- If the parent process of the receiving process is not executing a wait, the receiving process is made dormant.

- The parent process ID of each of the receiving process's existing child processes is set to 1. Dormant child processes are deleted.

- SIG_IGN -- The signal sig is to be ignored; the setting of func remains as SIG_IGN. Note that the signal SIGKILL cannot be ignored.

- function address -- Upon receipt of the signal sig, the receiving process is to execute the signal-catching function pointed to by func. The signal number sig is passed as the first argument to the signal-catching function; other arguments are unspecified.

  Upon return from the signal-catching function, the receiving process resumes from the point where it was when the signal was caught. The value of func for a caught signal is reset to SIG_DFL unless the catching function executes a call to the signal function to set it otherwise.

  Since the signal SIGKILL always causes process termination, its appearance in the signal function is not allowed.

RETURN VALUE:

Upon successful completion, signal returns the previous value of func for the specified signal sig. Otherwise, a value of -1 is returned and the variable errno is set to indicate the error.

DIAGNOSTICS:

The signal function fails if:

- The argument sig is an illegal signal number, including SIGKILL [EINVAL].

- The argument func points to an illegal address [EFAULT].

If a signal catcher is invoked while a process is executing a heap management function, and that signal catcher causes a recursive invocation of a heap management function by calling (even indirectly) any heap management function, the heap can be left in an inconsistent state. The heap can also be left in an inconsistent state if such a signal catcher abandons the heap management function using a nonlocal goto. The default signal catcher does neither of these things. (For the purpose of this note, the heap management functions are calloc, malloc, and free.)

RELATED FUNCTIONS:

kill, pause, setjmp, wait.

sin

Sine function.

FORMAT:

```
# include <math.h>

double sin (x)
double x;
```

ARGUMENTS:

x

Double-precision value.

DESCRIPTION:

The sin function returns the sine of a radian argument. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

RELATED FUNCTIONS:

acos, asin, atan, atan2, cos, tan.

sinh

Hyperbolic sine function.

FORMAT:

```
# include <math.h>

double sinh (x)
double x;
```

ARGUMENTS:

x

Double-precision value.

DESCRIPTION:

The sinh function computes the hyperbolic sine function for real arguments.

DIAGNOSTICS:

The sinh function returns a huge value of appropriate sign when the correct value would overflow.

RELATED FUNCTIONS:

cosh, tanh.

sleep

Suspend execution for interval.

FORMAT:

```
unsigned sleep (seconds)
unsigned seconds;
```

ARGUMENTS:

seconds

Number of seconds to suspend execution.

DESCRIPTION:

The sleep function suspends the current process from execution for a specified number of seconds. The actual suspension time may be less than that requested for two reasons: because scheduled wakeups occur at fixed 1-second intervals, and because any caught signal terminates the sleep following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by sleep is the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested sleep time, or premature arousal due to a caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling sleep; if the sleep time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred, and the caller's alarm catch routine is executed just before the sleep routine returns, but if the sleep time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening sleep.

DIAGNOSTICS:

If the sleep function is unable to set the alarm clock for any reason, the variables m4_errno and errno are set to indicate the reason and the hexadecimal value FFFF is returned. The first call to sleep may fail because the task is unable to obtain memory from the group work segment for the alarm clock or because it is unable to create the auxiliary task to listen for the alarm to go off. Reasons for failure are:

- Lack of group work segment memory--errno set to ENOMEM
- Lack of available LRN--errno set to AGAIN.

RELATED FUNCTIONS:

alarm, pause, signal.

smopen

Open file for storage management I/O.

FORMAT:

```
# include <stdio.h>

int smopen (path, oflag[, bsize])
unsigned char *path;
int oflag, bsize;
```

ARGUMENTS:

path

Pathname of file to be opened.

oflag

Access mode. Values are constructed by OR-ing flags from the following list:

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.
O_APPEND	Is allowed, but is ignored.
O_BINARY	Is allowed, but is ignored. The distinction between text and record mode accesses is meaningless at the storage management level.

bsize

Optional block size in characters. This is the maximum block size, not necessarily the actual block size. The default is 1024 characters.

DESCRIPTION:

The smopen function opens a file descriptor for the named file and sets the file status flags according to the value of oflag. The file is opened for storage management level I/O via the smread and smwrit functions.

The smopen function returns with the file pointer used to mark the current position in the file pointing to the first block of the file.

The new file descriptor is set to remain open across exec function calls.

No process can have more than 20 file descriptors open at once.

RETURN VALUE:

Upon successful completion, a nonnegative integer, the file descriptor is returned. Otherwise, errno and m4_errno are set to indicate the cause of failure and -1 is returned.

RELATED FUNCTIONS:

smread, smwrit.

smread

Read a block from a file.

FORMAT:

```
int smread (fildes, buf, nchars)
int fildes;
char *buf;
unsigned int nchars;
```

ARGUMENTS:

fildes

File descriptor obtained from an smopen function call.

buf

Buffer pointer.

nchars

Number of characters to read from file to buffer.

DESCRIPTION:

The smread function reads a block from the file associated with fildes into the buffer pointed to by buf. The block read is designated by the current value of the file's file pointer.

Upon successful completion, smread increments the file pointer by one block.

RETURN VALUE:

Upon successful completion a positive integer is returned indicating the size of the block read. If smread fails because the specified file is not open, errno and m4_errno are set to EBADF and 0x1800+EBADF, respectively, and -1 is returned. If smread fails for any other reason, errno is set to EFAULT, m4_errno is set to indicate the reason, and -1 is returned.

RELATED FUNCTIONS:

smopen, smwrit.

smwrit

Write a block to a file.

FORMAT:

```
int smwrit (fildes, buf, nchars)
int fildes;
char *buf;
unsigned int nchars;
```

ARGUMENTS:

fildes

File descriptor obtained from an smopen function call.

buf

Buffer address.

nchars

Number of characters to write from buffer to file.

DESCRIPTION:

The smwrit function writes a block of size nchars characters from the buffer pointed to by buf to the file associated with fildes. The file block written is the one designated by the current value of the file's file pointer.

Upon successful completion, smwrit increments the file pointer by one block.

RETURN VALUE:

Upon successful completion a positive integer is returned indicating the size of the block written.

DIAGNOSTICS:

If smwrit fails because the specified file is not open, errno and m4_errno are set to EBADF and 0x1800+EBADF, respectively, and -1 is returned. If smwrit fails for any other reason, errno is set to EFAULT, m4_errno is set to indicate the reason, and -1 is returned.

RELATED FUNCTIONS:

smopen, smread.

sprintf

Format output.

FORMAT:

```
# include <stdio.h>

int sprintf (s, format [, arg] ... )
char *s, format;
```

ARGUMENTS:

format

Format string.

arg

Optional argument to be printed.

s

Address of location to begin output.

DESCRIPTION:

The sprintf function places "output," followed by the null character (\0) in consecutive characters starting at *s; you must ensure that enough storage is available.

This function is equivalent to a call to fprintf, except that the argument s specifies an array into which the generated output is written instead of a file.

For more information on this function, refer to the description of printf.

RELATED FUNCTIONS:

ecvt, fprintf, printf, putc, scanf.

sqrt

Square root function.

FORMAT:

```
# include <math.h>

double sqrt (x)
double x;
```

ARGUMENTS:

x

Double-precision value.

DESCRIPTION:

The sqrt function returns the square root of x. X cannot be negative.

DIAGNOSTICS:

The sqrt function returns zero and sets errno to EDOM when x is negative.

RELATED FUNCTIONS:

exp, hypot, log, pow, sinh.

srand

Reset random number generator.

FORMAT:

```
srand (seed)
unsigned seed;
```

ARGUMENTS:

seed

Seed value.

DESCRIPTION:

The srand function reinitializes the random number generator function. It can be set to a random starting point by calling srand with any argument.

RELATED FUNCTIONS:

rand.

sscanf

Formatted input conversion.

FORMAT:

```
# include <stdio.h>

sscanf (s, format [,pointer]...)
char *s, *format;
```

ARGUMENTS:

s

Input character string.

format

Control string format.

pointer

Set of arguments indicating where the converted input should be stored.

DESCRIPTION:

The sscanf function reads from the character string s. This function reads characters, interprets them according to a format, and stores the results in its arguments. It requires a control string format and a set of optional pointer arguments indicating where the converted input should be stored.

The sscanf function is equivalent to a call to fscanf, except that the argument s specifies an array from which input is obtained rather than a file.

For more information on this function, refer to the description of fscanf.

RELATED FUNCTIONS:

atof, fscanf, getc, printf, scanf.

star_check

star_check

Validate star names.

FORMAT:

```
# include <star_name.h>

int star_check (star)
unsigned char *star;
```

ARGUMENTS:

star

Null-terminated string containing the star name to be validated.

DESCRIPTION:

The star_check function validates a star name to ensure that it has been formed according to the rules for constructing star names. For information on star names, see the Commands manual.

RETURN VALUE:

If star contains a validly constructed star name, one of the values STAR_NOT, STAR_SOME, or STAR_ALL is returned. STAR_NOT is returned when star is valid but is not a star name (does not contain asterisks or question marks). STAR_ALL is returned when star is a star name that matches every entry name (either **, *.**, or **.*). STAR_SOME is returned for all other valid star names.

If star does not contain a validly constructed star name, errno and m4_errno are set to EBADSTAR and 0x1800+EBADSTAR respectively and -1 is returned.

NOTE

For user convenience, all of the star name functions are declared in the <star_name.h> header file. The various STAR_... return values are also defined in this header file. EBADSTAR is defined in the <errno.h> header file.

RELATED FUNCTIONS:

star_match, star_name.

star_match

Validate and match star names.

FORMAT:

```
# include <star_name.h>

int star_match (star, source)
unsigned char *star, *source;
```

ARGUMENTS:

star

Null-terminated string containing the star name to be validated and matched with a source name.

source

Null-terminated string containing the entry name to be compared with the star name.

DESCRIPTION:

The star_match function implements the star convention by comparing an entry name with a name possibly containing stars or question marks (called a star name). Refer to the Commands manual for a description of the star convention and a definition of acceptable star name formats.

RETURN VALUE:

If star contains a validly constructed star name, either STAR_UNMATCH or STAR_MATCH is returned. STAR_UNMATCH is returned when star is valid but does not match source. STAR_MATCH is returned when star is valid and matches source.

If star does not contain a validly constructed star name, errno and m4_errno are set to EBADSTAR and 0x1800+EBADSTAR respectively and -1 is returned.

NOTE

Refer to the description of star_name to see how to list the directory entries that match a given star name.

RELATED FUNCTIONS:

star_check, star_name.

star_name

List directory entries matching star name.

FORMAT:

```
unsigned char *star_name (star, dir_path [, flags])
unsigned char *star, *dir_path;
[long flags;]
```

ARGUMENTS:

star

Null-terminated string containing the star name to be matched with the directory entries.

dir_path

Null-terminated string containing the pathname of the directory to be searched. If dir_path is null or points to a null string, the working directory is assumed.

flags

Optional bit pattern indicating which types of directory entries are to be considered for matching with the star name (see below). If flag is not present, all types of directory entries are considered.

DESCRIPTION:

The star_name function lists directory entries matching a star name. The function is called with a star name, a directory pathname, and an optional set of flags restricting the type of directory entries the star name is matched with. The directory is searched for all entries that match the star name and are not excluded by the flags. Information about these entries is returned in a "string of strings." For information on star names, see the Commands manual.

The value for flags is constructed by OR-ing values from the list below. The bit is set to cause the corresponding type of directory entry to be considered. Resetting a bit causes the corresponding type of directory entries to be ignored.

0x00000001	Sequential files that are members of a multivolume set but are not the last member in the set.
0x00000002	Sequential files that are either the last member of a multivolume set or are not a member of a multivolume set.
0x00000004	Relative files
0x00000008	Primary indexed files
0x00000010	Primary indexes
0x00000020	Alternate indexes
0x00000100	Dynamic files
0x00000200	Random files
0x00000400	IDS/II data base areas
0x00000800	Disk volumes
0x00001000	Fixed-relative files without deletable records
0x00002000	Directories
0x00004000	Links to pathnames
0x00008000	Fixed-relative files with deletable records
0x00020000	String-relative files

NOTE

The star_name function obtains memory for the string of strings containing its results from the heap via malloc and realloc. The caller is expected to return this memory to the heap via free.

RETURN VALUE:

If star_name is successful, it returns a pointer to a string of strings specifying the types and names of the matched directory entries. The star_name function can be successful and still match no names, in this case a null string of strings is returned. The string for each matched directory entry consists of a single character giving the type of the directory entry followed immediately by a null-terminated sequence of characters giving the directory entry's name.

The type characters are chosen from the following list:

'\000'	End of string of strings
'\001'	Sequential files that are members of a multivolume set but are not the last member in the set.
'\002'	Sequential files that are either the last member of a multivolume set or are not a member of a multivolume set.
'\003'	Relative files
'\004'	Primary indexed files
'\005'	Primary indexes
'\006'	Alternate indexes
'\011'	Dynamic files
'\012'	Random files
'\013'	IDS/II data base areas
'\014'	Disk volumes
'\015'	Fixed-relative files without deletable records
'\016'	Directories
'\017'	Links to pathnames
'\020'	Fixed-relative files with deletable records
'\022'	String-relative files

A null string of strings has the value: \000\000

DIAGNOSTICS:

If star_name fails, errno and m4_errno are set to indicate the reason and (unsigned char *)_0 is returned. The more common reasons for failure are:

- The dir_path argument does not name a directory [ENOTDIR].
- The star argument does not contain a validly constructed star name [EBADSTAR].
- Unable to obtain sufficient memory from the heap [ENOMEM].

Errors detected by open and read may also be encountered.

EXAMPLE:

If star is *.c and the matching directory entries are the sequential files foo.c and bar.c, star_name returns a pointer to the string of strings:

```
\002foo.c\000\002bar.c\000\000
```

RELATED FUNCTIONS:

star_check, star_match.

stat

Get file status.

FORMAT:

```
# include <types.h>
# include <stat.h>

int stat (path, buf)
char *path;
struct stat *buf;
```

ARGUMENTS:

path

File pathname. Read, write or execute access to the named file is not required, but all directories listed in the pathname leading to the file must be searchable.

buf

Pointer to a static structure into which information is placed concerning the file.

DESCRIPTION:

The stat function obtains information about the named file.

The contents of the structure pointed to by buf include the following members:

```
ushort   st_mode;   /*File mode                              */
ino_t    st_ino;    /*Inode number (N/A in MOD 400)          */
dev_t    st_dev;    /*ID of device containing                */
                    /*a directory entry for this file        */
dev_t    st_rdev;   /*ID of device                           */
                    /*This entry is defined only for         */
                    /*character special or block special
                      files                                  */
short    st_nlink;  /*Number of links (N/A in MOD 400)       */
ushort   st_uid;    /*User ID of the file's owner            */
ushort   st_gid;    /*Group ID of the file's group           */
off_t    st_size;   /*File size in characters (N/A)          */
time_t   st_atime;  /*Time of last access                    */
time_t   st_mtime;  /*Time of last data modification         */
                    /*Time measured in seconds since
                      00:00:00 GMT, Jan. 1, 1970             */
time_t   st_ctime;  /*Time of creation                       */
```

The st_atime member is the date/time when the file was last accessed. It is changed by the functions creat and read.

The st_mtime member is the date/time when the file was last modified. It is changed by the functions creat and write.

The st_ctime member is the date/time when the file was created. It is changed by the following functions: chown, creat, link, unlink, and write.

Information is not available in the members st ino, st_nlink, and st_size.

RETURN VALUE:

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and errno and m4_errno are set to indicate the error.

DIAGNOSTICS:

The stat function fails if:

- A component of the path prefix is not a directory [ENOTDIR].
- The named file does not exist [ENOENT].
- Search access is denied for a component of the path prefix [EACCES].

RELATED FUNCTIONS:

creat, fstat, link, stat, time, unlink.

strcat

Concatenate strings.

FORMAT:

```
char *strcat (s1, s2)
char *s1, *s2;
```

ARGUMENTS:

s_1, s_2

Null-terminated strings.

DESCRIPTION:

The strcat function appends a copy of string s_2 to the end of string s_1. It returns a pointer to the null-terminated result. This function does not check for overflow of any receiving string.

NOTE

All string movement is performed character by character, starting at the left. Thus overlapping moves toward the left work as expected, but overlapping moves to the right may not.

RELATED FUNCTIONS:

strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strtok.

strchr

Find character in string.

FORMAT:

```
char *strchr (s, c)
char *s, c;
```

ARGUMENTS:

s

String to search.

c

Character to seek.

DESCRIPTION:

The strchr function returns a pointer to the first occurrence of character c in string s, or NULL if c does not occur in the string. The null character terminating a string is considered to be part of the string.

The strchr function operates on null-terminated strings. This function does not check for overflow of any receiving string.

RELATED FUNCTIONS:

strcat, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strtok.

strcmp

Compare strings.

FORMAT:

```
int strcmp (s1, s2)
char *s1, *s2;
```

ARGUMENTS:

s_1, s_2

Null-terminated strings.

DESCRIPTION:

The strcmp function compares its arguments and returns an integer greater than, equal to, or less than zero, according to whether s_1 is lexicographically greater than, equal to, or less than s_2. This function does not check for overflow of any receiving string.

RELATED FUNCTIONS:

strcat, strchr, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strtok.

strcpy

Copy string.

FORMAT:

```
char *strcpy (s1, s2)
char *s1, *s2;
```

ARGUMENTS:

s_1, s_2

Null-terminated strings.

DESCRIPTION:

The strcpy function copies string s_2 to s_1, stopping after the null character has been moved. It returns s_1. This function does not check for overflow of any receiving string.

NOTE

All string movement is performed character by character, starting at the left. Thus overlapping moves toward the left work as expected, but overlapping moves to the right may not.

RELATED FUNCTIONS:

strcat, strchr, strcmp, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strtok.

strcspn

Substring operation.

FORMAT:

```
int strcspn (s1, s2)
char *s1, *s2
```

ARGUMENTS:

s_1, s_2

Null-terminated strings.

DESCRIPTION:

The strcspn function returns the length of the initial segment of string s_1 which consists entirely of characters not from string s_2. This function does not check for overflow of any receiving string.

RELATED FUNCTIONS:

strcat, strchr, strcmp, strcpy, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strtok.

strlen

Find length of string.

FORMAT:

```
int strlen (s)
char *s;
```

ARGUMENTS:

s

Null-terminated string.

DESCRIPTION:

The strlen function returns the number of non-null character in s. This function does not check for overflow of any receiving string.

RELATED FUNCTIONS:

strcat, strchr, strcmp, strcpy, strcspn, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strtok.

strncat

Concatenate portion of string.

FORMAT:

```
char *strncat (s1, s2, n)
char *s1, *s2;
int n;
```

ARGUMENTS:

s_1, s_2

Null-terminated strings.

DESCRIPTION:

The strcat function appends at most n characters of string s_2 to the end of string s_1. It returns a pointer to the null-terminated result. This function does not check for overflow of any receiving string.

NOTE

All string movement is performed character by character, starting at the left. Thus overlapping moves toward the left work as expected, but overlapping moves to the right may not.

RELATED FUNCTIONS:

strcat, strchr, strcmp, strcpy, strcspn, strlen, strncmp, strncpy, strpbrk, strrchr, strspn, strtok.

strncmp

Compare to portion of string.

FORMAT:

```
int strncmp (s1, s2, n)
char *s1, *s2;
int n;
```

ARGUMENTS:

s_1, s_2

Null-terminated strings.

n

Number of characters to check.

DESCRIPTION:

The strncmp function looks at up to n characters of string s_1 and compares it to argument s_2, and returns an integer greater than, equal to, or less than zero, according to whether s_1 is lexicographically greater than, equal to, or less than s_2. This function does not check for overflow of any receiving string.

RELATED FUNCTIONS:

strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncpy, strpbrk, strrchr, strspn, strtok.

strncpy

Copy n characters.

FORMAT:

```
char *strncpy (s1, s2, n)
char *s1, *s2;
int n;
```

ARGUMENTS:

s_1, s_2

Null-terminated strings.

n

Number of characters to copy.

DESCRIPTION:

The strncpy function copies exactly n characters of string s_2 to s_1, truncating or null-padding s_2; the target might not be null-terminated if the length of s_2 is n or more. It returns s_1. This function does not check for overflow of any receiving string.

NOTE

All string movement is performed character by character, starting at the left. Thus overlapping moves toward the left work as expected, but overlapping moves to the right may not.

RELATED FUNCTIONS:

strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strpbrk, strrchr, strspn, strtok.

strpbrk

Locate substring.

FORMAT:

```
char *strpbrk (s1, s2)
char *s1, *s2;
```

ARGUMENTS:

s_1, s_2

Null-terminated strings.

DESCRIPTION:

The strpbrk function returns a pointer to the first occurrence in string s_1 of any character from string s_2, or NULL if no character from s_2 exists in s_1. This function does not check for overflow of any receiving string.

RELATED FUNCTIONS:

strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strrchr, strspn, strtok.

strrchr

Find last occurrence of substring.

FORMAT:

```
char *strrchr (s, c)
char *s, c;
```

ARGUMENTS:

s

Null-terminated string.

c

Character to check for.

DESCRIPTION:

The strrchr function returns a pointer to the last occurrence of character c in string s, or NULL if c does not occur in the string. The null character terminating a string is considered to be part of the string. This function does not check for overflow of any receiving string.

RELATED FUNCTIONS:

strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strspn, strtok.

strspn

Get length of substring.

FORMAT:

```
int strspn (s1, s2)
char *s1, *s2;
```

ARGUMENTS:

s_1, s_2

Null-terminated strings.

DESCRIPTION:

The strspn function returns the length of the initial segment of string s_1 which consists entirely of characters from string s_2. This function does not check for overflow of any receiving string.

RELATED FUNCTIONS:

strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strtok.

strtok

String token operation.

FORMAT:

```
char *strtok (s1, s2)
char *s1, *s2;
```

ARGUMENTS:

s_1, s_2

Null-terminated strings.

DESCRIPTION:

The strtok function considers the string s_1 to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string s_2. The first call (with pointer s_1 specified) returns a pointer to the first character of the first token, and will have written a NULL character into s_1 immediately following the returned token. Subsequent calls with zero for the first argument work through the string s_1 in this way until no tokens remain. The separator string s_2 may be different from call to call. When no token remains in s_1, a NULL is returned. This function does not check for overflow of any receiving string.

NOTE

All string movement is performed character by character, starting at the left. Thus overlapping moves toward the left work as expected, but overlapping moves to the right may not.

RELATED FUNCTIONS:

strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn.

swab

Swap bytes.

FORMAT:

```
swab (fr, to, nbytes)
char *fr, *to;
int nbytes;
```

ARGUMENTS:

fr

Pointer to memory area from which bytes are taken.

to

Pointer to memory area in which bytes are placed.

nbytes

Number of bytes to move; argument should be an even number.

DESCRIPTION:

The swab function copies nbytes bytes pointed to by fr to the position specified by to, exchanging adjacent even and odd bytes.

This function is useful on machines where strings of characters are stored from right to left within words and from left to right from word to word, and where words are two characters wide. It is not particularly useful on DPS 6 machines.

sys_errlist

System error messages.

FORMAT

```
char *sys_errlist [];
```

ARGUMENTS:

None.

DESCRIPTION:

To simplify variant formatting of error messages, the vector of message strings sys_errlist is provided; the variable errno can be used as an index in this table to get the message string without the newline character. The variable sys_nerr is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

RELATED FUNCTIONS:

errno, perror, sys_nerr.

sys_nerr

Number of largest system error message.

FORMAT:

```
int sys_nerr;
char *sys_errlist [];
```

ARGUMENTS:

None.

DESCRIPTION:

To simplify variant formatting of messages, the vector of message strings sys_errlist is provided; the variable errno can be used as an index in this table to get the message string without the newline character. The variable sys_nerr is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

RELATED FUNCTIONS:

errno, perror, sys_errlist.

system

Issue a MOD 400 command.

FORMAT:

```
# include <stdio.h>

int system (string)
char *string;
```

ARGUMENTS:

string

Command line.

DESCRIPTION:

The system function causes the string to be given to MOD 400 as input as if the string had been typed as a command at a terminal. The current process waits until the command has completed, then returns the exit status of the command.

The MOD 400 command processor is used to process the string instead of /bin/sh. Command pathnames can be in MOD 400 or UNIX syntax. This function uses the search rules defined in the PATH environment line. The default PATH environment line specifies the referencing directory, the working directory, >>SYSLIB1, and >>SYSLIB2, in that order.

DIAGNOSTICS:

An exit status return of 127 is returned if the command processor could not be called successfully, and the variable m4_errno is set to indicate the reason.

tan

Tangent function.

FORMAT:

```
# include <math.h>

double tan (x)
double x;
```

ARGUMENTS:

x

Double-precision value.

DESCRIPTION:

The tan function returns the tangent of a radian argument. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

RELATED FUNCTIONS:

acos, asin, atan, atan2, cos, sin.

tanh

Hyperbolic tangent function.

FORMAT:

```
# include <math.h>

double tanh (x)
double x;
```

ARGUMENTS:

x

Double-precision value.

DESCRIPTION:

The tanh function computes the hyperbolic tangent function for real arguments.

RELATED FUNCTIONS:

cosh, sinh.

time

Get time.

FORMAT:

```
long time ((long *) 0)

long time (tloc)
long *tloc;
```

ARGUMENTS:

tloc

Pointer to memory area in which result is returned.

DESCRIPTION:

The time function returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

If tloc is not null, the return value is also stored in the location to which tloc points.

RETURN VALUE:

Upon successful completion, time returns the value of time. Otherwise, a value of -1 is returned, and the variable errno is set to indicate the error.

DIAGNOSTICS:

The time function fails if tloc points to an invalid address [EFAULT].

*

tmpnam

Create a name for a temporary file.

FORMAT:

```
# include <stdio.h>

char *tmpnam (s)
char *s;
```

ARGUMENTS:

s

Address of array to receive result.

DESCRIPTION:

The tmpnam function generates a file name that can safely be used for a temporary file. If (int)s is zero, tmpnam leaves its result in an internal static area and returns a pointer to that area. The next call to tmpnam destroys the contents of the area. If (int)s is nonzero, s is assumed to be the address of an array of at least L_tmpnam characters, where L_tmpnam is a constant defined in stdio.h; tmpnam places its result in that array and returns s as its value.

The tmpnam function generates a different file name each time it is called.

Files created using tmpnam and either fopen or creat are only temporary in the sense that they reside in a directory intended for temporary use, and their names are unique. You must use unlink to remove the file when its use is ended.

NOTES

1. If called more than 17,576 times in a single task group, tmpnam starts recycling previously used names.

2. Between the time a file name is created and the file is opened, it is possible for some other task group to create a file with the same name. This can never happen if that other task group is using tmpnam or mktemp, and the file names are chosen so as to render duplication by other means unlikely.

RELATED FUNCTIONS:

create, unlink, fopen, mktemp.

toascii

Character translation.

FORMAT:

```
# include <ctype.h>

int toascii (c)
int c;
```

ARGUMENTS:

c

Character to translate.

DESCRIPTION:

The toascii function translates a character into 7-bit ASCII.

The toascii function yields its argument with all bits turned off that are not part of a standard 7-bit ASCII character; it is intended for compatibility with other systems.

RELATED FUNCTIONS:

ctype, getc, toascii8, tolower, toupper.

toascii8

8-bit character translation.

FORMAT:

```
# include <ctype.h>

int toascii8 (c)
int c;
```

ARGUMENTS:

c

Character to translate.

DESCRIPTION:

The toascii8 function translates a character into 8-bit ASCII.

RELATED FUNCTIONS:

ctype, getc, toascii8, tolower, toupper.

tolower

Character translation.

FORMAT:

```
# include <ctype.h>

int tolower (c)
int c;
```

ARGUMENTS:

c

Character to translate.

DESCRIPTION:

The tolower function has as a domain all 8-bit ASCII codes (hexadecimal 0 through FF). If the argument represents an uppercase letter, the result is the corresponding lowercase letter. All other arguments in the domain are returned unchanged.

RELATED FUNCTIONS:

ctype, getc, toascii, toascii8, toupper.

_tolower

Character translation.

FORMAT:

```
# include <ctype.h>

int _tolower (c)
int c;
```

ARGUMENTS:

c

Character to translate.

DESCRIPTION:

The _tolower macrocall takes as an argument an uppercase letter. The result is the corresponding lowercase letter. All other arguments cause unspecified results.

RELATED FUNCTIONS:

ctype, getc, toascii, toascii8, toupper.

toupper

Character translation.

FORMAT:

```
# include <ctype.h>

int toupper (c)
int c;
```

ARGUMENTS:

c

Character to translate.

DESCRIPTION:

The toupper function has as a domain all 8-bit ASCII codes (hexadecimal 0 through FF). If the argument represents a lowercase letter that has a corresponding uppercase letter, the result is that uppercase letter. All other arguments in the domain are returned unchanged.

RELATED FUNCTIONS:

ctype, getc, toascii, toascii8, tolower.

_toupper

Character translation.

FORMAT:

```
# include <ctype.h>

int _toupper (c)
int c;
```

ARGUMENTS:

c

Character to translate.

DESCRIPTION:

The _toupper macrocall takes as an argument a lowercase letter that has a corrseponding uppercase letter. The result is the corresponding uppercase letter. All other arguments in the domain cause unspecified results.

RELATED FUNCTIONS:

ctype, getc, toascii, toascii8, tolower.

ttyname

Find name of a terminal.

FORMAT:

```
char *ttyname (fildes)
```

ARGUMENTS:

fildes

File descriptor of terminal.

DESCRIPTION:

The ttyname function returns a pointer to the null-terminated pathname of the terminal device associated with file descriptor fildes.

DIAGNOSTICS:

The ttyname function returns a null pointer (0) if fildes does not describe a terminal device.

tzset

Set time zone.

FORMAT:

```
void tzset ()
```

DESCRIPTION:

The tzset function sets the external variables timezone, daylight, and tzname, using either the external variable TZ (if present) or the system time zone. It is called by the asctime function, but you can also call it directly.

The value of TZ must be a time zone acronym, a time offset, and an optional daylight-savings time zone acronym.

- The time zone acronym is up to four characters long.
- The time offset represents the difference between local time in the designated time zone and GMT. The difference is represented by a string of digits with an optional leading minus sign (for locations east of Greenwich, England) and with an optional trailing .5 (for locations some odd number of half-hours from Greenwich).
- The optional daylight savings time zone acronym is up to four characters long.

For example, the setting for Boston would be EST5EDT.

RELATED FUNCTIONS:

asctime, ctime, gmtime, localtime, time; see also the list_stz and set_stz commands.

ucf_init, ucf_defc, ucf_defr, ucf_finish

Create a file.

FORMAT:

```
# include <ufas.h>

int ucf_init (path,org,[lrsz,cisz,iasz,grsz,mxsz])
unsigned char path[];
char org;
int lrsz, cisz, iasz, grsz, mxsz;

int ucf_defc (cmpl, cmp2)
int cmpl, cmp2;

int ucf_defr (rtype, [dup, ktype, ksize, kloc])
char ktype;
int rtype, dup, ksize, kloc;

int ucf_finish ( )
```

ARGUMENTS (for ucf_init):

path

File pathname.

org

File organization:

```
F_SEQ    -- Sequential
F_REL    -- Relative
F_IND    -- Indexed
F_DYN    -- Dynamic
F_CALC   -- Random
F_ALT    -- Alternate index
F_FIXREL -- Fixed-relative
F_STREL  -- String-relative
```

lrsz

For fixed-relative files, the logical record size, in characters; for other file organizations, the maximum record size, in characters. This number does not include record control information.

Default: Undefined for relative files; 216 characters for sequential, indexed, dynamic, and random files; 255 characters for string-relative files; and 256 characters for fixed-relative files.

cisz

Control interval size, in characters. This value must be a multiple of 256.

Default: For fixed-relative files, one physical sector; for other file organizations, 512 characters.

iasz

Initial allocation size. For sequential, relative, dynamic, random, and indexed files, the unit is control intervals; for fixed-relative files, the unit is records. You must specify either iasz or mxsz for random files.

Default: No initial allocation.

grsz

Size of additional space to be added to the file as it expands. For sequential, relative, dynamic, random, and indexed files, the unit is control intervals; for fixed-relative files, the unit is records.

Default: 40 physical sectors.

mxsz

Maximum file size. For sequential, relative, dynamic, random, and indexed files, the unit is control intervals; for fixed-relative files, the unit is records.

Default: No initial allocation.

ARGUMENTS (for ucf_defc):

cmpl, cmp2

For indexed files, cmpl is the number of free characters per control interval; cmp2 is the frequency of local overflow control intervals (for example, 10 means one in 10).

For random files, cmpl is the percentage of a data control interval that must be filled before inventory is updated (the default is 75 percent); cmp2 is the number of possible hash results, not more than the number of control intervals allocated to the file (the default is one per control interval).

For dynamic files, cmpl is the percentage of a data control interval that must be filled before inventory is updated (the default is 75 percent); cmp2 must be zero.

For alternate-index files, cmpl is the number of free characters per control interval; cmp2 must be zero.

ARGUMENTS (for ucf_defr):

rtype

A digit connecting a record descriptor to the record being processed, allowing different records in a file to have different record descriptors or key definitions. For indexed and alternate-index files, this argument must be zero.

dup

Duplicate key:

1 -- Duplicate keys allowed
0 -- Duplicate keys not allowed.

ktype

Key component data type:

BINARY -- Signed binary
CHARSTR -- Character string
DECIMAL -- Signed unpacked decimal
DECPCK -- Signed packed decimal
UDECPCK -- Unsigned packed decimal.

Specify a key type in uppercase for ascending key sequence; specify in lowercase for descending key sequence.

ksize

Size of the key field in characters.

kloc

Position of the first character of the key field within the record. (The first character of a record is number 1.)

DESCRIPTION:

The ucf_init, ucf_defc, ucf_defr, and ucf_finish functions create a file. You can create a sequential, relative, indexed, alternate index, dynamic, random, fixed-relative, or string-relative file. You must call ucf_init and ucf_finish to create a file; ucf_defc and ucf_defr are optional, depending on the file type. For more information on file creation, refer to the System Programmer's Guide--Volume I.

The ucf_init function constructs a create file descriptor. If you specify any of the optional arguments, you must provide a value for all of them, though you can specify zeros to take defaults. You must call this function first in the sequence.

The ucf_defc function collects more information needed for creating indexed, alternate index, random, or dynamic files. You must call ucf_init before calling this function.

The ucf_defr function is required for creating keyed files (indexed, alternate index, and random files). You must call ucf_init before calling this function. Call this function once for each key component or record type.

The ucf_finish function is required for all file types. You must call this function last in sequence.

RETURN VALUE:

Upon successful completion, these functions return a value of 0.

If these functions encounter an error, they set errno and m4_errno to indicate the error and return a value of -1.

uldiv

Divide unsigned long values.

FORMAT:

```
unsigned long uldiv (a, b)
unsigned long a, b;
```

ARGUMENTS:

a

Unsigned long dividend.

b

Unsigned long divisor.

DESCRIPTION:

The uldiv function performs division of the unsigned long value a by the unsigned long value b.

RELATED FUNCTIONS:

lgdiv, lgmul, lgrem, ulrem.

ulrem

Remainder function for unsigned long values.

FORMAT:

```
unsigned long ulrem (a, b)
unsigned long a, b;
```

ARGUMENTS:

a

Unsigned long dividend.

b

Unsigned long divisor.

DESCRIPTION:

The ulrem function returned the remainder function of a/b for unsigned long values.

RELATED FUNCTIONS:

lgdiv, lgmul, lgrem, uldiv.

umemchr

Locate character in memory.

FORMAT:

```
# include <memory.h>

unsigned char *umemchr (s, c, m)
unsigned char *s;
unsigned char c;
unsigned int m;
```

ARGUMENTS:

s

Pointer to memory area to check.

c

Character to seek.

m

Size of memory area in characters.

DESCRIPTION:

The umemchr function returns a pointer to the first occurrence of character c within the first m characters of memory area s, or (unsigned char *) 0 if c does not occur.

This function operates efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character).

NOTE

This function is declared in the <memory.h> header file.

RELATED FUNCTIONS:

memccpy, memchr, memcmp, memcpy, memset, umemcmp, umemcpy, umemset.

umemcmp

Memory-to-memory comparison.

FORMAT:

```
# include <memory.h>

int umemcmp (s1, s2, m)
unsigned char *s1, *s2;
unsigned int m;
```

ARGUMENTS:

s_1

First memory area to be compared.

s_2

Second memory area to be compared.

m

Size of memory area in characters.

DESCRIPTION:

The umemcmp function compares its arguments, looking at the first m characters only.

This function operates efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). It executes without a stack frame of its own, and it makes use of commercial instructions.

RETURN VALUE:

This function returns an integer less than, equal to, or greater than zero, depending on whether s_1 is less than, equal to, or greater than s_2. If m is zero, equality is indicated.

NOTES

1. This function is declared in the <memory.h> header file.

2. The umemcmp function uses 8-bit ASCII comparisons. Comparison proceeds from left to right until an unequal pair of characters is found or until all characters have been compared without finding an unequal pair. If an unequal pair is found, their ordering in the 8-bit ASCII code set determines the ordering of the two operands.

umemcpy

Memory-to-memory copy.

FORMAT:

```
# include <memory.h>

unsigned char *umemcpy (s1, s2, m)
unsigned char *s1, *s2;
unsigned int m;
```

ARGUMENTS:

s_1

Pointer to target memory area (output).

s_2

Pointer to source memory area (input).

m

Size of memory area in characters.

DESCRIPTION:

The umemcpy function copies the first m characters from memory area s_1 to s_2.

This function operates efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). This function does not check for the overflow of any receiving memory area. It executes without a stack frame of its own.

RETURN VALUE:

This function returns s_1.

NOTES

1. This function is declared in the <memory.h> header file.

2. The umemcpy function produces unspecified results if the memory areas overlap but are not identical.

umemset

Initialize memory.

FORMAT:

```
# include <memory.h>

unsigned char *umemset (s, c, m)
unsigned char *s;
unsigned char c;
unsigned int m;
```

ARGUMENTS:

s

Pointer to memory area to initialize.

c

Character to fill memory area.

m

Size of memory area in characters.

DESCRIPTION:

The umemset function sets the first m characters in memory area s to the value of character c.

This function operates efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). This function does not check for the overflow of any receiving memory area. It executes without a stack frame of its own, and makes use of commercial instructions.

RETURN VALUE:

This function returns *s.

NOTE

This function is declared in the <memory.h> header file.

RELATED FUNCTIONS:

memccpy, memchr, memcmp, memcpy, memset, umemchr, umemcmp, umemcpy.

ungetc

Push character back into input file.

FORMAT:

```
int ungetc (c, file)
char c;
FILE *file;
```

ARGUMENTS:

c

Character to push.

file

Pathname of input file.

DESCRIPTION:

The ungetc function pushes the character c back on an input file. That character is returned by the next getc call on that file. The ungetc function returns c.

One character of pushback is guaranteed provided something has been read from the file and the file is actually buffered. Attempts to push EOF are rejected.

DIAGNOSTICS:

The ungetc function returns EOF if it cannot push a character back.

RELATED FUNCTIONS:

getc, setbuf.

unlink

Remove directory entry.

FORMAT:

```
int unlink (path)
char *path;
```

ARGUMENTS:

path

Pathname of directory entry.

DESCRIPTION:

The unlink function deletes the file entry named by the path argument. If path is a link, the link is removed. If path is a file, the file is deleted.

RETURN VALUE:

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the variable errno is set to indicate the error.

DIAGNOSTICS:

The unlink function fails if:

- The volume is write protected [EROFS].
- The path argument points outside the task group's allocated address space [EFAULT].

RELATED FUNCTIONS:

close, link, open.

wait

Wait for event.

FORMAT:

```
int wait (stat_loc)
int *stat_loc;
int wait ((int *)0)
```

ARGUMENTS:

stat_loc

Pointer to memory area containing status information.

DESCRIPTION:

The wait function suspends the calling process until it receives a signal or, if a parent process, until one of its child processes terminates. If a child process terminates prior to the call on wait, there is an immediate return.

If stat_loc is not null, 16 bits of information called status are stored in the integer pointed to by stat_loc. The status argument can be used to differentiate between receipt of a signal and a terminated child process. If a child process terminates, status identifies the cause of termination and passes useful information to the parent. This is accomplished in the following manner:

- If the child process terminates due to an exit call, the low-order eight bits of status is zero and the high-order eight bits contain the low-order eight bits of the argument that the child process passed to exit.
- If the child process terminates due to a signal, the high-order eight bits of status is zero and the low-order eight bits contain the number of the signal that caused the termination. In addition, if a memory dump was produced, the hexadecimal value 0080 is moved into status.

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child is set to 1.

DIAGNOSTICS:

The first call to wait causes an interval timer to be created. If the process is unable to obtain memory from the group work segment for this timer, errno is set to ENOMEM, and -1 is returned.

The wait function fails and returns immediately if:

- The calling process has no existing child processes for which it is waiting [ECHILD].
- The pointer stat_loc refers to an illegal address [EFAULT].

RETURN VALUE:

If wait returns due to the receipt of a signal, a value of -1 is returned to the calling process, the variable errno is set to EINTR, and the variable m4_errno is set accordingly. If wait returns due to a terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and the variables errno and m4_errno are set to indicate the error.

RELATED FUNCTIONS:

exec family, exit, fork, pause, signal.

write

Write on a file.

FORMAT:

```
int write (fildes, buf, nchars)
int fildes;
char *buf;
unsigned nchars;
```

ARGUMENTS:

fildes

File descriptor obtained from a creat, dup, open, or pipe function.

buf

Address of buffer containing characters to be written.

nchars

Number of characters to write.

DESCRIPTION:

The write function attempts to write nchars characters from the buffer pointed to by buf to the file associated with the file descriptor fildes.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from write, the file pointer is incremented by the number of characters actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is unspecified.

If the O APPEND file status flag is set, the file pointer is set to the end of the file before each write.

If a write requests that more characters be written than there is room for (ULIMIT or the physical end of a medium), only as many characters as there is room for will be written. For example, if there is space for 20 characters more in a file reaching a limit, a write of 512 characters returns 20. The next write of a nonzero number of characters gives a failure return (except as noted below).

The write function does not allocate a buffer until it is needed. The function allocates 136-character buffers for the user-in, user-out, and error-out files and 512-character buffers for other files. The number of buffers ultimately allocated for a file is as follows:

- Binary files processed only by low-level I/O (read and write) get no buffers
- A user-in file processed only by low-level I/O gets no buffer
- String-relative files processed only by low-level I/O get no buffers
- All other files processed only by low-level I/O get one buffer each
- Files processed by high-level I/O get one more buffer than they would if processed only by low-level I/O.

RETURN VALUE:

Upon successful completion, the number of characters actually written is returned. Otherwise, -1 is returned and the variable errno is set to indicate the error.

DIAGNOSTICS:

The write function fails and the file pointer is unchanged if:

- The fildes argument is not a valid file descriptor open for writing [EBADF].
- An attempt was made to write a file that exceeds the task group's file size limit or the maximum file size [EFBIG].
- The buf argument points outside the task group's allocated address space [EFAULT].

RELATED FUNCTIONS:

creat, dup, open, pipe.

y0, y1, yn

Bessel functions.

FORMAT:

```
# include <math.h>

double y0 (x)
double x;

double y1 (x)
double x;

double yn, (n, x);
double x;
int n;
```

ARGUMENTS:

x

Double-precision value.

n

Order of Bessel function.

DESCRIPTION:

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders. The yn function returns the Bessel function of x of the second kind of order n; the value of x must be positive.

DIAGNOSTICS:

Zero and negative arguments cause y0, y1, and yn to return a huge negative value; the variable errno is set to EDOM.

RELATED FUNCTIONS:

j0, j1, jn.

Appendix A
C COMPILER DIAGNOSTIC MESSAGES

This appendix lists the C compiler diagnostic messages in alphabetical order. In messages, <---> indicates a variable.

Table A-1 lists C compiler error messages that pertain to the arguments of the M4_CC command. These messages are written to the user-out file. The messages fall into two categories:

W -- Warning error; the compiler discards the offending argument and continues

F -- Fatal error; the compiler terminates.

Table A-1. C Compiler Error Messages

Message	Class
Bad -SZ option (<--->)	W
-BU name <---> is too long	F
Can't execute <---> (errno=eeeee)	F
Fatal error in <---> (status=ssss, errno=eeeee)	F
Missing name after option <--->	F
Missing path after option <--->	F
Scientific Instruction Processor (SIP) required	F
Too many assembler options	W
Too many loader options	W
Too many preprocessor options	W
Try again	F
Unknown option <--->	F

Table A-2 lists the C compiler diagnostic messages. These messages are written to the error-out file and appear in this format:

prog_name: line lll: message

where prog_name is the name of the source unit containing the error, lll is the line number within the source unit, and message is the text of the message. The messages fall into three categories:

W -- Warning error; compilation continues

F -- Fatal error; compilation of the source unit terminates

O -- Optimization error; compilation continues with un-optimized assembly language.

Table A-2. C Compiler Diagnostic Messages

Message	Class
'? :' operator, types do not match across ':'	F
0-length row: <--->	F
<---> is not a register operand	O
<---> is not an entry point	O
<---> is not an opcode	O
<---> is not an option	O
<---> is not implemented	O
<---> may not have an initial value	F
<---> multiply defined	F
<---> must have an initial value	F
<---> operands missing	O
<---> redeclared	F
<---> undefined	F
<---> undefined; func. <--->	F
<--->: actuals too long	F
<--->: macro recursion	F
<--->: missing)	F
<--->: too many recursive calls	F
<--->: unterminated macro call	F
A CONST initializer in reentrant code cannot reference STATIC or EXTERN data	F
A static initializer cannot reference an AUTO	F
A static initializer cannot reference a REGISTER	F
A static initializer cannot reference a CONST in reentrant code	F
Ambiguous structure reference for <--->	F
Arg count	F

Table A-2 (cont). C Compiler Diagnostic Messages

Message	Class
Bad formal: <--->	F
Bad func. storage class	F
Bad include syntax	F
Bad structure/union/enum name	F
Bad type for field	F
Binary expression botch	F
Botch in outcode	F
Branch label too complex	O
Break/continue error	F
C0 internal error: tree not active	F
Call of non-function	F
Can't create <--->	F
Can't create work files	F
Can't find <--->	F
Can't find include file <--->	F
Can't read include file <--->	F
Cannot open <--->	O
Cannot open source file <--->	F
Cannot rewind workfile <--->	O
Case not in switch	F
Commercial instructions not implemented	O
Compiler botch: argument type	F
Compiler botch: call	F
Compiler botch: odd size	F
Compiler error (length)	F
Compiler error: all buffers in use	O
Compiler error: pname	F
Compiler error: too many active labels	O
Compound statement required	F
Conflict in storage class	F
Data cannot directly reference function <---> in reentrant code	F
Declaration syntax	F
Default not in switch	F
Disallowed conversion	F
Divide by zero	F
Divide check	F
Duplicate case (<--->)	F
Excessive -I file (<--->) ignored	F
Expression input botch	F
Expression overflow	F
Expression syntax	F
Extended Integer instructions not implemented	O
Extended Mode instructions not implemented	O
External definition syntax	F
Extraneous name <--->	F

Table A-2 (cont). C Compiler Diagnostic Messages

Message	Class
Floating %% not defined	F
Floating point size undefined	F
Freopen of stdin != stdin	F
Freopen of stdout != stdout	F
IF not implemented, true assumed	F
If-less else	F
If-less endif	F
Ignoring -D option (<--->)	F
Ignoring -U option (<--->)	F
Illegal #	F
Illegal character c in preprocessor if	F
Illegal character in <---> field	O
Illegal conditional	F
Illegal conversion	F
Illegal enum constant for <--->	F
Illegal enumeration <--->	F
Illegal indirection	F
Illegal initialization	F
Illegal lvalue	F
Illegal number <--->	F
Illegal operation on structure	F
Illegal operator in constant expression	F
Illegal register	O
Illegal storage class	F
Illegal structure operation	F
Illegal structure ref	F
Illegal type of operand	F
Illegal use of register	F
Illegal use of type	F
Illegal use of type name	F
Illegal use of void object	F
Inappropriate 'else'	F
Inappropriate parameters	F
Incompatible structures	F
Integer constant overflow, expression converted to long	F
Integer constant required	F
Intermediate file error (op=hhhh)	F
Long character constant	F
Lvalue required	F
Mask field missing	O
Masked and indexed bit instruction	O
Misplaced 'long'	F
Misplaced 'unsigned'	F
Missing '}'	F
More than 1 'default'	F

Table A-2 (cont). C Compiler Diagnostic Messages

Message	Class
Names <---> and <---> conflict	F
Negative field width	F
No END statement	O
No auto. aggregate initialization	F
No code table for op: <--->	F
No field initialization	F
No match for op <--->	F
No space	F
No strings in automatic	F
Nonterminated comment	F
Nonterminated string	F
Not an argument: <--->	F
Null dimension	F
Out of space	O
Out of space-- cl	F
Pow2 botch	F
Program too large	F
Rank too large	F
Register overflow: simplify expression	F
Required file name is missing	O
RESV out of place	O
Shift distance too large	F
Stack overflow botch	F
Statement syntax	F
Struct/union cited for <---> is undefined	F
Structure redeclaration	F
Switch table overflow	F
TITLE statement misplaced	O
TITLE statement missing	O
Token too long	F
Too many -D options, ignoring <--->	F
Too many -U options, ignoring <--->	F
Too many defines	F
Too many files	O
Too many formals: <--->	F
Too many initializers: <--->	F
Too many operands	O
Too many structure initializers	F
Too many structure members	F
Too many }'s	F
Too much declaring in an expression	F
Too much defining	F
Type clash	F
Type is too complicated	F

Table A-2 (cont). C Compiler Diagnostic Messages

Message	Class
Undefined control	F
Undefined structure	F
Undefined structure initialization	F
Unexpected EOF	F
Unexpected end of line	O
Unexpected end-of-file	O
Unimplemented field operator	F
Unimplemented structure assignment	F
Unknown character	F
Unknown flag <--->	F
Unknown keyword	F
Unreasonable include nesting	F
Warning: '&' requires lvalue but an array is not an lvalue, '&' ignored	W
Warning: <---> has zero length", cs->name	W
Warning: <---> may conflict with compiler generated labels	W
Warning: <---> redefined	W
Warning: <---> used for type punning	W
Warning: EQUAL operator, '==', may have been mistyped	W
Warning: Explicit 'extern' indicates declaration, definition accepted	W
Warning: c= operator assumed	W
Warning: char converted to unsigned char	W
Warning: char pointer converted to word pointer	W
Warning: extern int <---> implicitly declared	W
Warning: field may overflow	W
Warning: field may underflow	W
Warning: illegal macro name	W
Warning: incomplete qualification, all struct/union members named <---> offset	W
Warning: int converted to pointer	W
Warning: int converted to unsigned	W
Warning: module name <---> would cause assembly errors, default names are used	W
Warning: no struct/union cited	W
Warning: non-portable pointer operation	W
Warning: overflow in constant expression	W
Warning: pointer converted to int	W
Warning: very large data structure	W
Warning: zero length array	W
Write error on temp	F

Appendix B
ASCII CHARACTER SET

This appendix lists the 8-bit ASCII character set.

The control characters appearing in the 8-bit ASCII character set are defined as follows:

ACK	Acknowledge
BEL	Bell
BS	Backspace
CAN	Cancel
CR	Carriage Return
DC1	Device Control 1
DC2	Device Control 2
DC3	Device Control 3
DC4	Device Control 4
DEL	Delete
DLE	Data Link Escape
EM	End of Medium
ENQ	Enquiry
EOT	End of Transmission
ESC	Escape
ETB	End of Transmission Block
ETX	End of Text
SYN	Synchronous Idle
FF	Form Feed
FS	File Separator
GS	Group Separator
HT	Horizontal Tab
LF	Line Feed
LS0	Locking Shift 0
LS1	Locking Shift 1
NAK	Negative Acknowledgement
NUL	Null
RS	Record Separator
SOH	Start of Heading
STX	Start of Text
SUB	Substitute
US	Unit Separator
VT	Vertical Tab

The graphic characters in the 8-bit ASCII character set are defined as follows:

SP	Space
!	Exclamation Mark
"	Quotation Mark
#	Number Sign
$	Dollar Sign
%	Percent Sign
&	Ampersand
'	Apostrophe
(	Left Parenthesis
)	Right Parenthesis
*	Asterisk
+	Plus Sign
,	Comma
-	Minus Sign, Hyphen
.	Period, Decimal Point
/	Solidus, Slash
:	Colon
;	Semicolon
<	Less-than Sign
=	Equals Sign
>	Greater-than Sign
?	Question Mark
@	Commercial At Sign
[	Left Square Bracket
\	Reverse Solidus
]	Right Square Bracket
^	Circumflex Accent
_	Underline
`	Grave Accent
{	Left Curly Bracket
|	Vertical Line
}	Right Curly Bracket
~	Tilde
NBSP	No-Break Space
¡	Inverted Exclamation Mark
¢	Cent Sign
£	Pound Sign
¤	Currency Sign
¥	Yen Sign
¦	Broken Bar
§	Paragraph Sign, Section Sign
¨	Diaeresis, Umlaut
©	Copyright Sign
ª	Feminine Ordinal Indicator
«	Left Angle Quotation Mark
¬	Not Sign
SHY	Soft Hyphen
®	Registered Trade Mark Sign

¯	Macron, Overline, Overbar
°	Degree Sign
±	Plus-Minus Sign
²	Superscript Two
³	Superscript Three
´	Acute Accent
µ	Small Greek Letter Mu, Micro Sign
¶	Pilcrow (Paragraph Symbol)
·	Middle Dot
¸	Cedilla
¹	Superscript One
º	Masculine Ordinal Indicator
»	Right Angle Quotation Mark
¼	Vulgar Fraction One Quarter
½	Vulgar Fraction One Half
¾	Vulgar Fraction Three Quarters
¿	Inverted Question Mark
À	Capital A With Grave Accent
Á	Capital A With Acute Accent
Â	Capital A With Circumflex Accent
Ã	Capital A With Tilde
Ä	Capital A With Diaeresis
Å	Capital A With Ring Above
Æ	Capital Dipthong A with E
Ç	Capital C With Cedilla
È	Capital E With Grave Accent
É	Capital E With Acute Accent
Ê	Capital E With Circumflex Accent
Ë	Capital E With Diaeresis
Ì	Capital I With Grave Accent
Í	Capital I With Acute Accent
Î	Capital I With Circumflex Accent
Ï	Capital I With Diaeresis
Ð	Capital Icelandic Eth
Ñ	Capital N With Tilde

86-060

Ò	Capital O With Grave Accent
Ó	Capital O With Acute Accent
Ô	Capital O With Circumflex Accent
Õ	Capital O With Tilde
Ö	Capital O With Diaeresis
x	Multiplication Sign
Ø	Capital O With Oblique Stroke
Ù	Capital U With Grave Accent
Ú	Capital U With Acute Accent
Û	Capital U With Circumflex Accent
Ü	Capital U With Diaeresis
Ý	Capital Y With Acute Accent
Þ	Capital Icelandic Thorn
ß	Small German Sharp s
à	Small a With Grave Accent
á	Small a With Acute Accent
â	Small a With Circumflex Accent
ã	Small a With Tilde
ä	Small a With Diaeresis
å	Small a With Ring Above
æ	Small Dipthong a With e
ç	Small c With Cedilla
è	Small e With Grave Accent
é	Small e With Acute Accent
ê	Small e With Circumflex Accent
ë	Small e With Diaeresis
ì	Small i With Grave Accent
í	Small i With Acute Accent
î	Small i With Circumflex Accent
ï	Small i With Diaeresis
ð	Small Icelandic Eth
ñ	Small n With Tilde
ò	Small o With Grave Accent
ó	Small o With Acute Accent
ô	Small o With Circumflex Accent
õ	Small o With Tilde
ö	Small o With Diaeresis
÷	Division Sign
ø	Small o With Oblique Stroke
ù	Small u With Grave Accent
ú	Small u With Acute Accent
û	Small u With Circumflex Accent
ü	Small u With Diaeresis
ý	Small y With Acute Accent
þ	Small Icelandic Thorn
ÿ	Small y With Diaeresis

86-061

Table B-1. Eight-Bit ASCII Character Set

b4	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
b3	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
b2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
b1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

<table>
<tr><th>b4</th><th>b3</th><th>b2</th><th>b1</th><th>H2 \ H1</th><th>00</th><th>01</th><th>02</th><th>03</th><th>04</th><th>05</th><th>06</th><th>07</th><th>08</th><th>09</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>00</td><td>NUL</td><td>DLE</td><td>SP</td><td>0</td><td>@</td><td>P</td><td>`</td><td>p</td><td></td><td></td><td>NBSP</td><td>°</td><td>À</td><td>Đ</td><td>à</td><td>ð</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>01</td><td>SOH</td><td>DC1</td><td>!</td><td>1</td><td>A</td><td>Q</td><td>a</td><td>q</td><td></td><td></td><td>¡</td><td>±</td><td>Á</td><td>Ñ</td><td>á</td><td>ñ</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>02</td><td>STX</td><td>DC2</td><td>"</td><td>2</td><td>B</td><td>R</td><td>b</td><td>r</td><td></td><td></td><td>¢</td><td>²</td><td>Â</td><td>Ò</td><td>â</td><td>ò</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>1</td><td>03</td><td>ETX</td><td>DC3</td><td>#</td><td>3</td><td>C</td><td>S</td><td>c</td><td>s</td><td></td><td></td><td>£</td><td>³</td><td>Ã</td><td>Ó</td><td>ã</td><td>ó</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>04</td><td>EOT</td><td>DC4</td><td>$</td><td>4</td><td>D</td><td>T</td><td>d</td><td>t</td><td></td><td></td><td>¤</td><td>´</td><td>Ä</td><td>Ô</td><td>ä</td><td>ô</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>05</td><td>ENQ</td><td>NAK</td><td>%</td><td>5</td><td>E</td><td>U</td><td>e</td><td>u</td><td></td><td></td><td>¥</td><td>µ</td><td>Å</td><td>Õ</td><td>å</td><td>õ</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>06</td><td>ACK</td><td>SYN</td><td>&</td><td>6</td><td>F</td><td>V</td><td>f</td><td>v</td><td></td><td></td><td>¦</td><td>¶</td><td>Æ</td><td>Ö</td><td>æ</td><td>ö</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>07</td><td>BEL</td><td>ETB</td><td>'</td><td>7</td><td>G</td><td>W</td><td>g</td><td>w</td><td></td><td></td><td>§</td><td>•</td><td>Ç</td><td>×</td><td>ç</td><td>÷</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>08</td><td>BS</td><td>CAN</td><td>(</td><td>8</td><td>H</td><td>X</td><td>h</td><td>x</td><td></td><td></td><td>¨</td><td>¸</td><td>È</td><td>Ø</td><td>è</td><td>ø</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>09</td><td>HT</td><td>EM</td><td>)</td><td>9</td><td>I</td><td>Y</td><td>i</td><td>y</td><td></td><td></td><td>©</td><td>¹</td><td>É</td><td>Ù</td><td>é</td><td>ù</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>10</td><td>LF</td><td>SUB</td><td>*</td><td>:</td><td>J</td><td>Z</td><td>j</td><td>z</td><td></td><td></td><td>ª</td><td>º</td><td>Ê</td><td>Ú</td><td>ê</td><td>ú</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td><td>11</td><td>VT</td><td>ESC</td><td>+</td><td>;</td><td>K</td><td>[</td><td>k</td><td>{</td><td></td><td></td><td>«</td><td>»</td><td>Ë</td><td>Û</td><td>ë</td><td>û</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>0</td><td>12</td><td>FF</td><td>FS</td><td>,</td><td><</td><td>L</td><td>\</td><td>l</td><td>|</td><td></td><td></td><td>¬</td><td>¼</td><td>Ì</td><td>Ü</td><td>ì</td><td>ü</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>13</td><td>CR</td><td>GS</td><td>-</td><td>=</td><td>M</td><td>]</td><td>m</td><td>}</td><td></td><td></td><td>SHY</td><td>½</td><td>Í</td><td>Ý</td><td>í</td><td>ý</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>14</td><td>LS1</td><td>RS</td><td>.</td><td>></td><td>N</td><td>^</td><td>n</td><td>~</td><td></td><td></td><td>®</td><td>¾</td><td>Î</td><td>Þ</td><td>î</td><td>þ</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>15</td><td>LS0</td><td>US</td><td>/</td><td>?</td><td>O</td><td>_</td><td>o</td><td>DEL</td><td></td><td></td><td>¯</td><td>¿</td><td>Ï</td><td>ß</td><td>ï</td><td>ÿ</td></tr>
</table>

86-004

GLOSSARY

byte

A DPS 6 byte is eight bits long. In this manual, the terms byte and character are synonymous.

character

In this manual, the terms character and byte are synonymous.

character array

A sequence of characters.

control terminal

See standard file.

dot

At the beginning of a pathname, this specifies the current working directory, equivalent to the UNIX link . (period). See link.

dot-dot

The UNIX link .. (period period), referring to the immediately superior directory. The MOD 400 equivalent is <. See link.

effective group ID

This concept applies to UNIX, not MOD 400. An active process has an effective user ID and an effective group ID that are used to determine file access rights. The effective user ID and effective group ID are equal to the task's real user ID and real group ID, respectively, unless the task or one of its ancestors evolved from a file that had the set-user-ID bit or set-group-ID bit set. These bits do not exist in, and are not set by, MOD 400.

effective user ID

See effective group ID.

file

File names consisting of up to 12 characters (in contrast to 14 characters in UNIX) are allowed to name an ordinary file, special file, or directory. The MOD 400 file naming conventions are listed in the MOD 400 Concepts manual.

file access

Read, write, and execute search rights on a file are granted to a process in accordance with MOD 400 access control.

file descriptor

An integer from 0 to 19 that designates a file to be processed by low-level I/O. See low-level I/O.

group

Each user is a member of a group, corresponding to the MOD 400 account. The group is identified by a group name, equivalent to the MOD 400 account ID, and by a positive integer called the real group ID (which has no MOD 400 equivalent). An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

group name

See group.

group work segment (GWS)

In MOD 400, the area of memory from which Get Memory system service macrocalls obtains memory.

heap

An area of memory from which the functions malloc and calloc obtain storage.

high-level I/O

Functions (such as fopen and fprint) that return a pointer to a file. See low-level I/O.

initial user-in file

The first file designated as user-in. Generally, this is an interactive terminal or a batch input file.

initial user-out file

The first file designated as user-out. Generally, this is an interactive terminal or a batch output file.

link

A UNIX directory entry. By convention, a UNIX directory contains at least two links, . and .., referred to as dot and dot-dot, respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

login name

See user.

low-level I/O

Functions (such as close, open, read, and write) that use file descriptors. See high-level I/O.

null character (NUL)

The ASCII character 00. In C, it is represented as \0.

null pathname

Unless specifically stated otherwise, the null pathname is treated as if it named a nonexistent file.

null pointer

The value obtained by casting 0 into a pointer. This value never matches any legitimate pointer, so many functions that return pointers will return a null pointer to indicate an error.

parent process ID

The parent process ID of a process is the process ID of its creator.

pathname

A null-terminated character string starting with an optional slash, followed by zero or more directory names separated by slashes, optionally followed by a file name. If a pathname begins with a slash, the path search begins at the root directory. Otherwise, the search begins from the current working directory. A slash by itself names the root directory.

process

A process corresponds to a MOD 400 task. Each active process in the system is uniquely identified by a positive integer called a process ID. A new process is created by a currently active process.

process ID

The process ID is derived from the address of the MOD 400 task control block as follows:

```
pid=(int)(address of TCB>>5)
```

process group

Each active process is a member of a process group. The process group corresponds to the MOD 400 task group. Each process group is identified by a positive integer called the process group ID. This grouping permits signaling to related processes.

process group ID

The process group ID is derived from the two-character task group ID of MOD 400 as follows:

```
(int)(task group ID)
```

real group ID

See group.

real user ID

See user.

referencing directory

The directory in which the program's bound unit was found.

root directory

Each process has associated with it a root directory and a current working directory for the purpose of resolving pathname searches. A process's root directory need not be the root directory of the root file system.

search rules

A list of directories MOD 400 examines to locate a file in: (1) exec calls to file names, (2) system calls to a simple pathname, and (3) all references to bound units. The search rules are:

1. Any directories in a PATH environment variable
2. The referencing directory
3. The working directory
4. >SYSLIB1
5. >SYSLIB2

Refer to the description of the find_file function.

signal catcher

A function invoked when a specified signal is received by the function's process. Refer to the description of the signal function.

standard file

The standard input file stdin corresponds to the MOD 400 file user-in. The standard output file stdout corresponds to the MOD 400 file user-out. The standard error file stderr corresponds to the MOD 400 file error-out. The control terminal is equivalent to the MOD 400 initial file command-in.

stderr file

See standard file.

stdin file

See standard file.

stdout file

See standard file.

string

A sequence of characters ending with a null character.

terminal

The concept of a terminal is the same in UNIX and MOD 400. The ttyname is equivalent to the MOD 400 symbolic peripheral device name of the terminal (for example, !TTY12). The control terminal is equivalent to the MOD 400 initial command-in file, while the terminal ID is equivalent to the MOD 400 pathname of the initial command-in file.

terminal ID

See terminal.

ttyname

See terminal.

user

The concept of a user is the same under UNIX and MOD 400. Each user allowed on the system is identified by a login name, equivalent to the MOD 400 person ID, and by a positive integer called a real user ID (which has no MOD 400 equivalent). Under MOD 400, the login name can be up to 12 characters in length. It is possible for several users to share a single user ID; however, they have unique login names. Multiple logins are also supported.

INDEX

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	DPS-6 GCOS 6 MOD 400 C USER'S GUIDE	ORDER NO.	CW35-02
		DATED	March 1986

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ____________________ DATE ____________

TITLE ____________________

COMPANY ____________________

ADDRESS ____________________

____

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	DPS 6 GCOS 6 MOD 400 C USER'S GUIDE	ORDER NO.	CW35-02
		DATED	March 1986

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ____________________ DATE ____________

TITLE ____________________

COMPANY ____________________

ADDRESS ____________________

____

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell