SWTPc 'C' Compiler

for the UniFLEX Operating System

Remastered 2023 by Jacob Beeksma
V1.1

SWTPc 'C' Compiler Manual

UniFLEX release 22.1

The SWTPc 'C' compiler is the product of James Mccosh. SWTPc is indebted to Mr. McCosh for the excellence of his work. The 'C' compiler system provides an excellent professional tool for operating systems software, utilities, and intricate data processing applications.

All questions regarding the SWTPc 'C' compiler system should be directed to:

Southwest Technical Products
21 West Rhapsody
San Antonio, Texas 78233

attn: 'C' compiler support group

Table of Contents

Compiler introduction

NAME

cc - compile from the 'C' programming language to Motorola M6809 machine code

SYNTAX

cc [flags] file…

DESCRIPTION

One or more files may be compiled together. Cc manages the compilation through up ·to four stages: preprocessor, compilation to assembler code, assembly to relocatable module and loading to binary. The binary output file may be no-text, shared-text or a relocatable module.

OPERATION

The compiler·accepts three types of source files, provided each name on the command line has the relevant·postfix:

".c" for a C source file,
".a" foran assembly language source file,
".r" for a relocatable module.

There are two modes of operation: multiple source file and single source file. The compiler selects the mode by inspecting the command line. The usual mode is single source and is specified by having only one source file name on the command line. Of course, more than one source file may be compiled together by using the '#include' facility in the source code. In this mode, the compiler will use the name obtained by removing the postfix from the name supplied on the command line, and the output will have this name. For example,

++ cc foo.c

will leave an executable file called 'foo' in the current directory.

The multiple source mode is specified by having more than one source file name on the command line. In this mode, the object code output file will have the name 'output' in the current directory. Also, the relocatable modules generated as intermediate files will be left in the same directories as their corresponding source files with the postfixes changed to ".r". For example,

++ cc foo1.c /fred/foo2.c

will leave an executable file called 'output' in the current directory, one called 'foo1.r' in the current directory and 'foo2.r' in '/fred'.

FLAGS

Cc recognizes several fiags that modify the compilation process if needed. All flags are recognized before compilation commences so they may be placed anywhere on the command line. As is usual under UniFLEX, flags may be run together as in '+lt'. The exception is '+o=<filename>'. A '-' (minus sign) may be used in place of the '+' in any of the following.

+a suppresses assembly, leaving the output as assembler code in a file whose name is postfixed '.a'.

+r suppresses loading and linking library·modules into an executable program. Output left in files with postfixes '.r'.

+m in multiple source mode, combine the relocatable output modules into one withoμt searching the standard library to resolve external references.

+t generate a shared-text output file. The default_ is a no-tex output file. See the UniFLEX 'link-edit'. documentation for explanations of these terms.

+o=<filename> over-rides the above output file naming. The output file will be left with <filename> as its name. This flag does not make sense in multiple source mode and either the +a or +r flag is also present.

+L the source code is output as comments with the assembler code.

+n no assembly code is produced by the compiler; an aid to checking the syntax of a source file.

+S stops the generation of stack-checking code. This flag should be used with great care when the application is extremely time-critical and when the use of the stack by compiler generated code is fully understood.

The following table gives example command lines that show the use of some of the flags available:

command line	Action	Output files
cc foo.c	compile an executable program	foo
cc foo.c +a	compile to assembly code	foo.a
cc foo.c +r	compile to relocatable module	foo.r
cc foo[1-3].c	compile to executable program	foo1.r, foo2.r, foo4.r, output
cc foo1.c foo2.a	compile foo1.c, assemble foo2.a and combine into an executable program	foo1.r, foo2.r, output
cc foo[1-3].c +a	compile to assembly code	foo1.a, foo2.a, foo3.a
cc foo[1-3].c +to=foo	compile to an executable shared-text program	foo

LANGUAGE

The source language of cc is proper subset of C as described in 'The C Programming Language' by Kernighan and Ritchie (hereafter referred to as K & R). Exceptions are as follows:

Bit fields are not yet supported.

Assignment operators such as '+=', '-=' etc. are supported, but only in this new form; not '=+' or '=-'.

"#ifdef (or #ifndef) …[#else…] #end if" is supported but "#if <constant expression>" is not. These constructs may not be nested.

It is not possible to extend macro definitions or strings over more than one line of source code.

The escape sequence for new-line '\n' does not indicate line-feed (hex 0A), but rather carriage-return (hex OD). This is because UniFLEX use the latter for all end-of-line indications. All programs which use '\n' for end-of-line (which includes all programs in K & R) will work unaltered as intended. See below for a way to express the line-feed character.

LANGUAGE EXTENSIONS

A line beginning "#asm" switches the compiler into a mode which passes all subsequent lines not beginning with "#endasm" unchanged to the output. This is a useful facility for inserting lines of assembly code. The "#endasm" switches the mode back to normal.

Care should be exercised when using this directive that the correct code segment is adhered to. Normal code from the compiler is in the text segment and if your assembly code uses the data segment, be sure to put a 'text' directive at the end to leave the state correct for following compiler generated code.

The escape sequences for non-printing characters in character constants and strings (see K & R page 181) are extended as follows:

line-feed LF \l

This is to distinguish the ASCII line feed character (hex 0a) from '\n' which is the same as '\r' (hex Od) on UniFLEX.

bit pattern ***

This is used to allow any valid octal, decimal, or hexadecimal 8-bit value to be transmitted directly. This is useful for carriage control and terminal control characters on output files. Examples of octal, decimal, and hexadecimal bit patterns follow:

```
\377    -- octal value corresponding to the decimal value 255
\d255   -- the decimal value 255
\xff    -- hexadecimal value corresponding to the decimal value 255
```

OBJECT SIZES

Each type of 'variable or constant requires a specific amount of central memory. The sizes of the basic types in bytes are as follows:

char	1
int, unsigned	2
pointer	2
long	4
float	4
double	8

CC follows the PDP11 implementation in that characters are converted to integers by sign extension, 'short' or 'short int' means int, 'long int' means long and '1ong float' means double.

REGISTER VARIABLES

Up to two register variables may be declared in a function. The only types permitted for register variables are int, unsigned and pointer. Invalid register variable declarations are ignored ; i.e. the storage class is made auto (see K & R page 81).

A considerable saving in code size and speed can be made by judicious use of register variables. The most efficient use of register variables is for pointers and counters for loops.

SYSTEM CALLS

The system interface supports almost all the system calls of UniFLEX, however, in order to facilitate the portability of programs from other operating systems, some of the names of the calls are different from the UniFLEX names. The details of the calls and a name cross-reference are provided in 'C System Calls'.

STANDARD LIBRARY

It is essential to head any source file which uses functions from the standard 1ibrary with '#include <stdio.h>. See 'C Standard Library' for details.

PLEASE NOTE that if output via printf(), fprintf() or sprintf() **of long integers** is required, the source MUST call pflinit() at some point; this s necessary so that programs not involving longs do not have the extra longs output code appended. Similarly if floats or doubles are to be printed, 'pffinit()' MUST be called.

WARNINGS

Because the target machine for this compiler, the 6809, is an 8/16 bit processor, it has been possible to generate reasonably efficient code for 8 and 16 bit objects. However, code for 32 and 64 bit values can at best be four times longer and slower. The moral is: don't use a long, float or double where an int or unsigned will do.

A number of temporary files are created in the directory '/tmp' during compilation and it is important to ensure that enough space is available on the relevant disc drive. As a rough guide, at least three times the number of blocks in the largest source file should be free.

The identifiers 'etext', 'edata' and 'end' are pre-defined in the loader and may be used to establish the addresses of the end of executable text, initialized data and un-initialized data respectively. The method of using them is to take the address using the '&' operator.

e.g. foo(&end);

The important thing to remember is that these identifiers may not be re-defined or re-declared.

FILES

Note that the UniFLEX relocating assembler and the UniFLEX linking loader required by cc are the TSC relasmb and link-edit programs which may be obtained with cc from SWTPc. Compilation and/or execution of a C language program with cc requires the following UniFLEX files.

/bin/cc	compiler executive program
/usr/bin/c.prep	macro pre-processor
/usr/bin/c.comp	compiler proper
/bin/relasmb	UniFLEX relocating assembler
/usr/bin/c.load	linking loader
/bin/link-edit	UniFLEX linking loader (used for the '+m' option)
/lib/clib.l	standard library, arithmetic routines and system interface
/usr/include	standard directory for '#include' files; any file included using '<' and '>' instead of quotes will be searched for here

SEE ALSO

C Standard Library
C System Calls
relasmb
link-edit

The C Programming Language
by Brian W. Kernighan and Dennis M. Ritchie,
Published by Prentice-Hall

DIAGNOSTICS

Cc provides reasonably good error detection, but as with most structured language compilers, one syntax error is likely to destroy the neat structure of a program, and cascade further errors. If the number of errors becomes too large (currently more than 30), the compiler aborts the program. Diagnostics from the compiler are routed to the standard output and may be re-directed or piped.

BUGS

Cc has been used internally by SWTPc for some time and at the time of distribution all known bugs have been removed. However, a problem report has been placed at the end of this manual and should be used to report bugs or problems noted while using cc.

NOTES

C Standard Library

INTRODUCTION

The Standard Library is a collection of fundamental functions which cover higher-level I/0 and computational or convenience routines.

The higher-level I/O functions provide facilities normally considered part of the definition of other languages; for example the FORMAT 'statement' of Fortran. In addition automatic buffering of I/0 channels is carried out which considerably improves the speed of file access as fewer system calls are necessary.

The functions in this category should not be confused with the lower-level system calls with similar names. Nor should 'file pointers' be confused with 'file descriptors'. The standard library functions maintain a structure for each file open holding status information and a pointer into its buffer. A user program uses a pointer to this structure as the 'identity' of the file (which is provided by 'fopen()') and passes it to the various I/0 functions. They in turn will make the lower-level system calls required when necessary.

USING A FILE POINTER IN A SYSTEM CALL OR A FILE DESCRIPTOR IN A STANDARD LIBRARY CALL is a common mistake among beginners to C and if made, will be sure to CRASH YOUR PROGRAM.

The convenience functions include facilities for copying, comparing and concatenating strings, making numbers out of strings and doing the donkey work in accessing system information such as the time or the password file.

In the pages which follow, the functions available are described in terms of what they do and the parameters they expect. The 'USAGE' section in each shows the name of the function and the type returned (if not int). The declaration of arguments is shown as it would be written in the function definition to indicate the types expected by the function. If it is necessary to include a file this is shown in the 'USAGE' section by '#include <filename>'.

Most of the header files required to be included must reside in the directory '/usr/include'. If the file is included in the source program using angle bracket delimiters instead of the usual double quotes, the compiler will append this path name to the file name. For example, '#include <stdio.h>' is equivalent to '#/include "/usr/include/stdio.h"'.

Please note that if the type of the value returned by a function is not int, you should make a pre-declaration in your program before calling it. For example, if you wish to use 'atof()', you should predeclare it by having the statement 'double atof();' somewhere in your program before a call to it. Some functions which have associated header files in '/usr/include' which should be included will be pre-declared for you in the header. An example of this is 'ftell()' which is pre-declared in 'stdio.h'. If you are in any doubt, read the header file.

atof, atoi, atol
ASCII to number conversions

USAGE

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

long atol(ptr)
char *ptr;

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

DESCRIPTION

Conversions of the string pointed to by 'ptr' to the relevant number type are carried out by these functions. They cease to convert a number when the first unrecognized character is encountered.

Each skip leading spaces and tab characters. Atof() then recognizes an optional sign followed by a digit string possibly containing a decimal point, then an optional 'e' or 'E', an optional sign and a digit string. Atol() and atoi() recognize an optional sign and a digit string.

BUGS

Overflow causes unpredictable indications. There are no error indications if this occurs.

crpass
encrypt a password

USAGE

```
char *crpass(strl, str2, "aa")
char *strl, str2[16];
```

DESCRIPTION

Crpass returns an encrypted password. "str1" is a pointer to an 8 character string containing raw data (the unencrypted password). The encrypted password is returned in the 16 character string, "str2", which is returned as the value of crpass. The last argumemt is a pointer to a string and must be two lower·case alphabetics. Crpass does not look for null terminators in either "strl" or "str2".

SEE ALSO

getpass()

ctime, localtime, asctime
return details of date and time

USAGE

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

#include <time.h>

struct tm *localtime(ptr)
long *ptr;

char *asctime(tptr)
struct tm *tptr;
```

DESCRIPTION

Localtime returns a pointer to a structure containing details of the time as expressed in the long integer pointed to by 'ptr'. The value pointed to by 'ptr' should normally be obtained by using the time() system call (q.v.).

The structure declared in /usr/include/time.h (which must be included, as above, in any program calling localtime() or asctime,() is as follows:

```
struct tm {
   int   tm_secs;    /* second */
   int   tm_min;     /* minute */
   int   tm_hour;    /* hour on a 24 hour clock */
   int   tm_mday;    /* day of the month */
   int   tm_mon;     /* month */
   int   tm_year;    /* year - 1900 */
   int   tm_wday;    /* day of the week */
   int   tm_yday;    /* day of the year */
};
```

Asctime takes as argument a pointer to a structure such as the above and returns a pointer to a string showing the time in the same format as the UniFLEX 'date' command. The string will be 26 characters long, including a new-line and a nul1, and in the following form:
10:14:50 Thu Nov 19 1981\n\0.

Ctime uses the long integer pointed to by 'ptr' to return a pointer to a string in the same form as that of asctime.

SEE ALSO

System call time()

WARNINGS

The returned pointers point to static areas of store. This means that to preserve the information in the string or structure over subsequent calls you must copy it to another place or extract the information•

feof, ferror, clearerr, fileno
return status information of files

USAGE

```
#include <stdio.h>

feof(fp)
FILE *fp;

ferror(fp)
FILE *fp;

clearerr(fp)
FILE *fp;

fileno(fp)
FILE *fp;
```

DESCRIPTION

Feof returns non-zero if the file associated with 'fp' has reached its end, zero otherwise.

Ferror returns on-zero if an error condition has arisen on access to the file 'fp', zero otherwise. The error condition persists, preventing further access to the file by other Standard Library functions, until either the file is closed or it is cleared by clearerr.

Clearerr resets the error condition on the file 'fp'. This does NOT 'fix' the file or prevent the error from occurring again; it merely allows Standard Library functions at least to try.

Fileno returns the file descriptor nunber of the file 'fp'. A -1 is returned for an invalid call.

WARNINGS

These functions are actually macros defined in '/usr/include/stdto.h' so their names cannot be redeclared.

SEE ALSO

System call open()
fopen()

fflush, fclose
flush or close a file

USAGE

```
#include <stdio.h>

fflush(fp)    I
FILE *fp;

fclose(fp)
FILE *fp;
```

DESCRIPTION

Fflush causes a buffer associated with the file pointer 'fp' to be cleared by writing out to the file; of course only if the file was opened for write or update. It is not normally necessary to call fflush but it can be useful when, for example, normal output is to 'stdout' and it is wished to send something to 'stderr' which is un-buffered. If fflush were not used and 'stdout' referred to the terminal, the 'stderr' message will appear before large chunks of the 'stdout' message even though the latter was 'written' first.

Fclose calls fflush to clear out the buffer associated with 'fp', closes the file and frees the buffer for use by another 'fopen' call.

The exit() system call and normal termination of a program causes fclose to be called for each open file

ERRORS

EOF is returned if 'fp' does not refer to an output file or there is an error on writing to to the file.

SEE ALSO

System call close()
fopen(), setbuf()

findstr, findnstr
string search

USAGE

```
findstr(pos, string, pattern)
int pos;
char *string, *pattern;

findnstr(pos, string, pattern, size)
int pos, size;
char *string, *pattern;
```

DESCRIPTION

These functions search the string pointed to by 'string' for the first instance of the pattern pointed to by 'pattern' starting at position 'pos' (where the first position is 1 not 0). The returned value is the position of the first matched character of the pattern in the string or zero if not found.

Findstr stops searching the string when a null byte is found in 'string'.

Findnstr only stops searching at position 'pos' + 'len' so it may continue past null bytes.

BUGS

The current implementation does not use the most efficient algorithm for pattern matching so that use on very long strings is likely to be somewhat slower than it might be.

SEE ALSO

index(), rindex()

fopen
open a file and return a file pointer

USAGE

```
#include <stdio.h>

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

DESCRIPTION 1

Fopen returns a pointer to a file structure (file pointer) if the file named in the string pointed to by 'filename' can be validly opened with the action in the string pointed to by 'action'.

The valid actions when opening a file are "r" for read, "w" for write and "a" for append.

Opening for write will perform a 'creat()', that is if a file with the same name exists it will be truncated to zero length. Append means open, for write and position to the end of the file. Writes to the file via 'putc()' etc. will extend the file. Only if the file does not already exist will it be created.

Note that the type of a file structure is pre-defined in 'stdio.h' as FILE so that a user program may declare or define a file pointer by, for example ' FILE *f; '.

Three file pointers are available and can be considered open the moment the program runs:

```
stdin   the standard input     - equivalent to file descriptor 0,
stdout  the standard output    - equivalent to file descriptor 1,
stderr  standard error output  - equivalent to file descriptor 2.
```

All files are automatically buffered, except stderr, unless made unbuffered by a call to setbuf() (q.v.).

WARNINGS

The 'action' passed as an argument to fopen must be a pointer to a string; NOT a character. For example,

```
fp = fopen("fred","r"); is correct but
fp = fopen("fred",'r'); is not.
```

ERRORS

Fopen returns NULL (0) if the call was unsuccessful.

SEE ALSO

System call open()
fclose()

fread, fwrite
read/write binary data

USAGE

```
#include <stdio.h>

fread(ptr, size, number, fp)
int ptr, size, number;
FILE *fp;

fwrite(ptr, size, number, fp)
int ptr, size, number;
FILE *fp;
```

DESCRIPTION

Fread reads from the file pointed to by 'fp', 'number' items of size, 'size' into memory starting at 'ptr'. The best way to pass the argument 'size' to fread is by using 'sizeof'. This function returns the number of items actually read.

Fwrite writes to the file pointed at by 'fp', 'number' items of size, 'size' reading them from memory starting at 'ptr'.

ERRORS

Both functions return 0 at end of file or error.

SEE ALSO

System calls read() and write()
fopen(), getc(), putc(), printf()

fseek, rewind, ftell
position in a file or report current position

USAGE

```
#include <stdio.h>

fseek(fp, offset, place)
int place;
FILE *fp;
long offset;

rewind(fp)
FILE *fp;

long ftell(fp)
FILE *fp;
```

DESCRIPTION

Fseek re-positions the next character position of a file for either read or write. The new position is at 'offset' bytes from the beginning if 'place' is 0, the current position if 1 or the end if 2. Fseek sorts out the special problems of buffering. Note that using 'lseek()' on a buffered file will produce unpredictible results.

Rewind is equivalent to 'fseek(fp, 01, 0)'.

Ftell returns the position of the next character to be read from or written to a file.

ERRORS

Fseek returns -1 if the call is invalid.

SEE ALSO

System call lseek()

gcd, lgcd
greatest common divisor

USAGE

```
gcd(argl, arg2);
int argl, arg2;

long lgcd(argl, arg2};
long argl, arg2;
```

DESCRIPTION

Gcd calculates the greatest common divisor of two integers (arg1, arg2). The method used is an adaptation of Euclid's algorithm, except that zero is returned if either argument is zero.

Lgcd is the 'long' version of gcd.

ERRORS

Gcd and lgcd return a 0 if the call is invalid.

getc, getchar, getw
return character or word to be read from a file

USAGE

```
#include <stdio.h>

getc(fp)
FILE *fp;

getchar()

getw(fp)
FILE *fp;
```

DESCRIPTION

Getc returns the next character from the file pointed to by 'fp'.

Getchar is equivalent to 'getc(stdin)'.

Getw returns the next word from the file pointed to by 'fp'.

ERRORS

A -1 is returned for end of file.

SEE ALSO

putc(), fread(), (open(), gets(), ungetc()

getpass
read a password

USAGE

```
char *getpass(prompt);
char *prompt;
```

DESCRIPTION

Getpass reads 'a password from the standard error file (usually connected to the user's terminal), after prompting with the null-terminated string "prompt" and disabling echoing. If the prompt is NULL, no prompt 1s displayed. A pointer is returned to a null-terminated string of at most 8 characters.

FILES

stderr

SEE ALSO

crpass()

WARNINGS
The return value points to static data whose content is overwritten by each call.

getpw, getpwuid, getpwnam, setpwent, endpwent
get password entry

USAGE

```
#include <pwd.h

struct passwd *getpwent()

struct passwd *getpwuid(uid)
int uid;

struct passwd *getpwnam(name)
char *name;

setpwent ();

endpwent ();
```

DESCRIPTION

These functions provide mechanisms for accessing the system password file (/etc/log/password) and breaking out the entry fields into a structure.

Getpwent, getpwuid and getpwnam each return a pointer to a structure defined in the include file as follows:

```
struct passwd {     /*header for access to the password file */
  char *pw_name;    /* user log in name * /
  char *pw_passwd;  /* encrypted password*/
  int   pw_uid;     /* user id */
  char *pw_dir;     /* home directory*/
  char *pw_shell;   /* log in program (his shell) */
};
```

See the UniFLEX manual under 'System Manager's Guide' for an explanation of these fields.

Getpwent reads the next line in the password file (opening it if necessary), setpwent rewinds to the beginning of the file and endpwent closes it.

Getpwuid searches from the beginning of the file until a matching user id is found, getpwnam does the same but tries to match the user log-in name.

ERRORS

A null pointer (0) is returned on end-of-file or error.

WARNINGS

The information is left in static storage so must be copied out if required to be preserved over more than one call.

gets, fgets
input a string.

USAGE

```
#include <stdio.h>

char *gets(s)
char *s;

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

DESCRIPTION

Fgets reads characters from the file 'fp' and places them in the buffer pointed to by 's' up to a carriage return ('\n') but not more than 'n'-1 characters. A null character is appended to the end of the string.

Gets is similar to fgets applied to 'stdin' but no maximum is stipulated and the '\n' is replaced by a null.

Both functions return their first arguments.

ERRORS·

Both functions return NULL on end of file or error.

BUGS

The different treatment of the '\n' by these functions is retained here for portability reasons.

SEE ALSO

puts(), getc(), scanf(), fread()

isalpha, isupper, islower, isdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii
character classification

USAGE

```
#include <ctype ,h>.

isalpha (c);

etc.
```

DESCRIPTION

These use table look-up to classify characters according to their ascii value. The header file defines them as macros which means that they are implemented as fast, in-line code rather than subroutine. Each results in non-zero for true or zero for false.

The correct value is guaranteed for all integer values in isascii but the result is unpredictable in the others if the argument is outside the range -1 to 127.

The truth tested by each is as follows:

isalpha	c is a letter
isdigit	c is a digit
isupper	c is an upper case letter
islower	c is a lower case letter
isalnum	c is a letter or a digit
isspace	c is a space, tab character, newline, carriage return or formfeed
iscntrl	c is a control character (0 to 32) or DEL (127)
ispunct	c is neither control nor alpha-numeric
isprint	c is printable (32 to 126)
isascii	c is in the range -1 to 127

l3tol, ltol3
convert between long integers and 3-byte integers.

USAGE

```
l3tol(lp, cp, n)
int n;
long *lp;
char *cp;

ltol3(cp, lp, n)
int n;
long *lp;
char *cp;
```

DESCRIPTION

Certain system values, such as disc addresses, are maintained in three-byte form rather than four-byte; these functions enable arithmetic to be used on them.

L3tol converts a vector of 'n' three-byte integers pointed to by 'cp' into a vector of long integers starting at 'lp'. Ltol3 does the opposite.

mktemp
create unique temporary file name.

USAGE

```
char *mktemp(name)
char *name;
```

DESCRIPION

Mktemp may be used to ensure that the name of a temporary file is unique in the system and so does not clash with any other file name.

'Name' must point to a string whose last five characters are 'X'; the X's will be replaced with the ascii representation of the task id.

For example, if 'name' points to "foo.XXXXX" and the task id is 351 the returned value points at the same place in storage but it now holds
"foo.35111"

SEE ALSO

System call getpid()

perror
system error message

USAGE

```
perror(s)
char *s;
```

DESCRIPTION

Perror produces a short error message on the standard error file describing the last error encountered during a call to the system from a C program.

printf, fprintf, sprintf
formatted output

USAGE

```
#include <stdio.h>
printf(control [,arg0[,argl..]])
char *control;

fprintf(fp, control [,arg0[,argl..]])
FILE *fp;
char *control;

sprintf(string, control [,arg0[,arg1.. ]])
string [];
char *control;
```

DESCRIPTION

These three functions are used to place numbers and strings on the output in formatted, human readable form.

Fprintf places its output on the file 'fp', printf on the standard output and sprintf in the buffer pointed to by 'string'. Note that it is the user's responsibility to ensure that this buffer is large enough.

The 'control' string determines the format, type and number of following arguments expected by the function. If the control does not match the arguments correctly, the results are unpredictable.

The control may contain characters to be copied directly to the output and/or format specifications. Each format specification causes the function to take the next successive argument for output.

A format specification consists of a '%' character followed by (in this order) :

An optional minus sign ('-') meaning left justification in the field.

An optional string of digits indicating the field width required. The field will be at least this wide and may be wider if the conversion requires it. The field will be padded on the left unless the above minus sign is present, in which case it will be padded on the right. The padding character is, by default, a space but if the digit string starts with a zero ('0') it will be '0'.

An optional dot ('.') and a digit string, the precision, which for floating point arguments indicates the number of digits to follow the decimal point on conversion and for strings the maximum number of characters from· the string argument that are to be printed.·

An optional character 'l' indicating that the following 'd','x' or 'o' is the specification of a long integer _argument

A conversion character which shows the type of the argument and the desired conversion. The recognized conversion characters are:

d,o,x The argument is an integer and the conversion is to decimal, octal or hexadecimal respectively.

u The argument is an integer and the conversion is to unsigned decimal in the range O to 65535.

f The argument is a double and the form of the conversion is '[-]nnn.nnn' where the digits after the decimal point are specified as above. If not specified, the precision defaults to six digits. If the precision is 0, no decimal point or following digits are printed.

e The argument is a double and the form of the conversion is '[-]n.nnn(+or-)nn'; one digit before the decimal point and the precision controls the number following.

g The argument is a double and either the 'f' format or the 'e' format is chosen, whichever is the shortest.

Note in each of the above double-conversions, the last digit is rounded.

c The argument is a character.

s The argument is a pointer to a string. Characters from the string are printed up to a null character or until the number of characters "indicated by the precision have been printed. If the precision is O or missing, the characters are not counted.

% No argument corresponding; '%' is printed.

WARNINGS

Normally, all output via the standard library function is buffered except to the standard error output unless disabled using 'setbuf()'. For convenience, printf (not fprintf) has been implemented to call 'fflush()' before returning to the user program so that output to the terminal is more immediate.

SEE ALSO

Kernighan & Ritchie pages 145-147
putc(), scanf()

putc, putchar, putw
put character in a file

USAGE

```
#include <stdio.h>

putc(ch, fp)
char ch;
FILE *fp;

putchar(ch);
char *ch;

putw(w, fp)
int w;
FILE *fp;
```

DESCRIPTION

Putc adds the character 'ch' to the file 'fp' at the current writing position and advances the position indicator.

Putchar is implemented as a macro (defined in the header file) and is equivalent to 'putc(ch, stdout)'.

Putw appends lword 'w' to the file 'fp' at the current writing position and advances the position indicator.

Output via putc is normally buffered except when the buffering is disabled by 'setbuf()' or output is to the standard error file.

ERRORS

These functions return the character argument from a successful call and EOF on end-of-file or error.

SEE ALSO

open(), fclose(), flush(), getc(), puts(), printf(), fread()

puts, fputs
put a string on a file

USAGE

```
#include <stdio.h>
puts(s)
char *s;

fputs(s, fp)
char *s;
FILE *fp;
```

DESCRIPTION

Fputs copies the (null-terminated) string pointed to by 's' onto the file 'fp'.

Puts copies the string 's' onto the standard output and appends '\n'.

The terminating null is not copied by either function.

WARNINGS

The inconsistency of the new-line being appended by puts and not by fputs is dictated by history and the desire for compatibility.

qsort
quick sort

USAGE

```
qsort(base, n, size, compfunc)
char *base;
int (*compfunc)();
/* which means a pointer to a function returning an int */
```

DESCRIPTION

Qsort implements the quick-sort algorithm for sorting an arbitrary array of items.

'Base' is the address of the array of 'n' items of size 'size'. 'Compfunc' is a pointer to a comparison routine supplied by the user. It will be called by qsort with two pointers to items in the array for comparison and should return an integer -which is less than, equal to or greater than O where respectively the first item is less than, equal to or greater than the second.

rand, srand, rrand
random number generator

USAGE

```
srand(seed)
int seed;

rrand{)

rand()
```

DESCRIPTION

Rand uses a 32-bit M-Sequence random number generator with period 2**32-1 to return successive pseudo-random numbers in the range from 0 to 2**15-1. Notice that this is the maximum positive short integer.

The generator is re-initialized by calling srand with 1 as its argument. Calling srand with an argument not equal to 1 will initialize the generator using the argument as a seed.

Calling rrand will re-seed the generator using the current date and time of day (in seconds).

scanf, fscanf, sscanf
input string interpretation·

USAGE

```
#include <stdio.h>

fscanf(fp, control [,pointer…])
FILE *fp;
char *control;

scanf(control[,pointer…])
char *control;

sscanf(string, control[,pointer…])
char *string , *control;
```

DESCRIPTION

These functions perform input formatting and conversion in a manner that is compatible with the output function 'printf()'.

Fscanf performs ·conversions from the file 'fp', scanf from the standard input and sscanf from the string pointed to by 'string'.

Each function expects a control string containing conversion specifications and zero or more pointers to objects into which the converted values are stored.

The control string may contain three types of field: spaces, tab characters or '\n' each of which match any of the three in the input; characters not among the above and not '%' which must match characters in the input; a '%' followed by an optional '*' indicating suppression of ·assignment, an optional field width maximum and a conversion character indicating the type expected.

A conversion character controls the conversion to be applied to the next field and indicates the type of the corresponding pointer argument. A field consists of consecutive non-space characters and ends at either a character inappropriate for the conversion or when a specified field width is exhausted. When one field is finished, white-space characters are passed over until the next field is found.

The following conversion characters are recognized:

d A decimal string is to be converted to an integer.

o An octal string; the corresponding argument should point to an integer.

s A string of non-space characters is expected and will be copied to the buffer pointed at by the corresponding argument and a null ('\0') is appended. The user must ensure that the buffer is large enough. The input string is considered terminated by a space, tab or ('\n').

x A hexadecimal string for conversion to an integer.

c A character is expected and is copied into the byte pointed to by the argument. The white-space skipping is suppressed for this conversion. If a field width is given, the argument is assumed to point to a character array and the number of characters indicated is copied to it. To ensure that the next non-white-space character is read, use '%ls' and point the argument at two bytes.

e,f A floating point representation is expected on the input and the argument must be a pointer to a float. Any of the usual ways of writing floating point numbers are recognized.

[This denotes the start of a set of match characters the inclusion or exclusion of which delimits the input field. The white-space skipping is suppressed. The corresponding argument should be a pointer to a character array. If the first character in the match string is not '^' charactere are copied from the input as long as they can be found in the match string. If the first character in the match string is '^' the copying continues while characters cannot be found in the match string. The match string is delimited by a ']'.

D,O,X Similar to d,o,x above but the corresponding argument is considered to point to a long integer.

E,F Similar to e,f above but the corresponding argument should point to a double.

% A match for'%' is sought; no conversion takes place.

Each of these functions returns a count of the number of fields successfully matched and assigned.

ERRORS

These funtions return EOF on end of input or error and a count which is shorter than expected for unexpected or unmatched items.

BUGS

The return counnt of matches/assignments does not include character matches and assignments suppressed by '*'.

WARNINGS

The argunents must ALL be pointers. It is a common error to call scanf with the value of an item rather than a pointer to it.

SEE ALSO

atoi(), atof{), getc{), printf()
Kernighan and Ritchie pages 147-150

setbuf
fix file buffer

USAGE

```
#include <stdio .h>

setbuf(fp, buffer)
FILE *fp;
char *buffer;
```

DESCRIPTION

When the first character is written to or read from a file after it has been opened by 'fopen()', a buffer is obtained from the system if required and assigned to it. Setbuf may be used to forestall assigning a user buffer to the file.

Setbuf must be used after the file has been opened and before any I/0 has taken place.

The buffer must be of sufficient size and a value for a manifest constant, BUFSIZ, is defined in the header file for use in declarations.

If the 'buffer'argument is NULL (0), the file becomes un-buffered and characters are read or written singly.

Note that the standard error output is normally unbuffered ·and, on UniFLEX, the standard output is buffered. The latter may change if and when new hardware improves the speed of single-character terminal writing.

SEE ALSO

fopen(), getc(), putc()

setjmp, longjmp
non-local goto

USAGE

```
#include <setjmp.h>

setjmp(env)
jmp-buf env;

longjmp(env, val)
jmp_buf env;
```

DESCRIPTION

Setjmp saves its stack environment in 'env' for later use by longjmp.

Longjmp restores the stack environment saved by the last call of setjmp. Longjmp returns in such a way that execution continues as if the call of setjmp had just returned the value 'val' to the function that invoked setjmp.

sleep
stop execution for a time

USAGE

```
sleep(seconds)
unsigned seconds;
```

DESCRIPTION

The current task is stopped for the specified time. The actual time may be up to one second less than specified because the system schedules wakeups at fixed one·second intervals of real time.

Implementation 1s by issuing an 'alarm()' call and pausing until the interrupt occurs. Any previous state of the 'alarm clock' for the task is saved and restored before continuing. If the time requested of sleep is longer than the time to the next alarm interrupt, the task sleeps until that interrupt would have occurred and the interrupt is sent at the next second.

SEE ALSO

System calls alarm() and pause()

strcat, strcmp, strcpy, strlen, index, rindex
string functions

USAGE

```
char *strcat{sl, s2)
char *sl, *s2;

strcmp(sl, s2)
char *sl, *s2;

char *strcpy(sl, s2)
char *sl, *s2;

strlen(s)
char *s;

char */ndex(s, ch)
char *s, ch;

char *rindex'(s, ch)
char *s, ch;
```

DESCRIPTION

These functions perfom high-level string manipulation. All strings passed to these functions are assumed null-terminated.

Strcat appennds a copy of the string pointed to by 's2' to the end of the string pointed to by 's1'.

Strcmp compares strings 'sl' and 's2' for lexicographic order and returns an integer less than, equal to or greater than O where, respectively, 'sl' is less than, equal to or greater than 's2'.

Strcpy copies characters from 's2' to the space pointed to by 'sl' up to and including the null byte.

Strlen returps the number of non-null characters in 's'.

Index returns a pointer to the first occurrence of 'ch' in 's' or NULL if not found.

Rindex returns a pointer to the last occurrence of 'ch' in 's' or NULL if not found.

WARNINGS

Strcat and strcpy have no means of checking that the space provided is large enough. It 1s the user's responsibility to ensure that string space does not overflow.

SEE ALSO

findstr()

system
shell command request

USAGE

```
system(string)
char *string;
```

DESCRIPTION

System passes its argument to 'shell' which executes it as a command line. The task is suspended until the shell command is completed and system returns the shell's exit status.

SEE ALSO

System calls exec() and wait()
UniFLEX command shell

ttyname, isatty
find name of a terminal

USAGE

```
char *ttyname(fd)
int fd;

isatty(fd)
int fd;
```

DESCRIPTION

Ttyname returns a pointer to the null-terminated path name of the terminal device associated with the file descriptor 'fd'.

Isatty returns '1' if 'fd' is associated with a terminal device and 0 otherwise.

ERRORS

Ttyname returns a '0' if 'fd' does not describe a terminal device.

ungetc
put character back on input

USAGE

```
#include <stdio,.h>

ungetc(ch, fp)
char ch;
FILE *fp;
```

DESCRIPTION

This function alters the state of the input file buffer such that the next call of 'getc()' returns 'ch'.

Only one character may be pushed back and at least one character must have been read from the file.

'Fseek()' erases any pushback.

ERRORS

Ungetc returns its character argument unless no pushback could occur, in which case EOF is returned.

SEE ALSO

getc(), fseek()

C System Calls

INTRODUCTION

This section of the C compiler manual is a guide to the system calls available from C programs.

It is NOT intended as a definitive description of UniFLEX system calls as these are adequately described in the UniFLEX manual. However, for most calls enough information is available here to enable the programmer to write system calls into programs without looking further.

All system calls are implemented by means of interface routines in the library which are automatically loaded at compile time. This allows a certain amount of latitude to the implementer of the compiler and has enabled some calls not available directly to an assembly language programmer.

The names used for the system calls and parameter values are chosen so that programs transported from other machines or operating systems should compile and run with as little modification as possible. However, care should be taken as the parameters and returned values of some calls may not be compatible to those on other systems. Those already familiar with UniFLEX names and values should take particular care. Some calls do not share the same name as the UniFLEX assembly-language equivalents. The assembly-language equivalent call is shown, where there is one, on the relevant page of the C call description and a cross-reference list is provided for those already familiar with UniFLEX calls.

The normal error indication on return from a system call is a returned value of '-1'. The relevant error will be found in the pre-defined int 'errno'. This location always contains the error from the last erroneous system call. Definitions for the errors for inclusion in the program are in '/usr/include/errno.h'.

In the 'SEE ALSO' sections on the following pages, unless otherwise stated, the references are to other system calls.

Where '#include' files are shown, it is not mandatory to include them but it might be convenient to use the manifest constants defined in them rather than integers. It certainly makes for more readable programs.

abort
stop the program and produce a core dump

USAGE

abort()

ASSEMBLER EQUIVALENT

swi

DESCRIPTION

This cal1 causes an EMT trap (swi) which aborts the program and causes a memory image to be written out to the file 'core' in the current directory.

WARNINGS

Please note that this call is implemented, like all the others, as a call to a library subroutine which in turn executes the 'swi'. This means that the address of the call in the user's program will be on the stack at the moment the core dump is made and not in the program counter.

access
give file accessibility

USAGE

```
access(fname, perm)
int perm;
char *fname;
```

ASSEMBLER EQUIVALENT

```
sys chacc, fname, perm
```

DESCRIPTION

Access returns 0 if the access modes specified in 'perm' would be granted if an attempt were made to access the file named by 'fname', '-1' otherwise.

The allowed values for 'perm' are any combination of

4 read,
2 write,
1 execute and
0 directory path search and file existence.

ERRORS

The appropriate error indication, if a value of '-1' is returned, may be found in 'errno'.

SEEE ALSO

stat

alarm
cause delayed interrupt

USAGE

```
alarm(seconds)
```

ASSEMBLER EQUIVALENT

```
<seconds in D>
sys alarm
<previous seconds in D>
```

DESCRIPTION

Alarm causes an SIGALRM (10) interrupt to be sent to the task after 'seconds' have elapsed.

The returned value is the amount of time left in the 'alarm clock' before the call.

SEE ALSO

pause(), signal()
library function sleep()

brk, sbrk, cdata
request additional working memory

USAGE

```
brk(address)

cdata(address)

char *sbrk(increase)
```

ASSEMBLER EQUIVALENT

```
sys break, address

sys cdata, address
```

DESCRIPTION

Break requests that the program's addressable data area should include 'address'.

Sbrk requests that the current available data area be increased by 'increase' bytes and returns a pointer to the lowest address in the the newly available memory.

Memory acquired by calls to brk or sbrk is logically contiguous with that from the last such call.

Cdata is the same as brk except that an attempt is made to make the memory acquired be physically contiguous. This is necessary for providing memory for use by DMA devices.

When a program starts execution, the highest available address is 'end'-1 ; i.e. the last byte in the data storage area. If reference is made to addresses above this and below the stack pointer, there will be upredictable results; quite probably a system crash.

Subsequent calls to sbrk raise the lower limit of this 'no-go' area making available extra working memory when required.

ERRORS

Brk returns 0 if the break could be set at the requested place, -1 otherwise. Sbrk also returns -1 on error. Cdata returns -1 if the amount of contiguous physical memory is unavailable.

SEE ALSO

UniFLEX shel1 command chd

chdir
change directory

USAGE

```
chdir(dirname)
char *dirname;
```

ASSEKBLER EQUIVALENT

```
sys chdir, dirname
```

DESCRIPTION

This call changes the the current working directory for the running task. 'dirname' is a pointer to a string giving a pathname for a directory.

ERRORS

Chdir returns O after a successful call, -1 if 'dirname' is not a directory path name or it is not searchable.

chmod
change access permissions of a file.

USAGE

```
#include <modes.h>

chmod(fname,perm)
int perm;
char *fname;
```

ASSEMBLER EQUIVALENT

```
sys chprm,fname,perm
```

DESCRIPTION

Chmod changes the permission bits associated with a file. 'fname' must be a pointer to a file name and 'perm' should contain the desired bit pattern.

The allowable bit patterns are defined in the include file as follows

```
/*permissions*/
#define S_IPRM      0xff  /* mask for permission bits*/
#define S_IREAD     0x01  /* owner read */
#define S_IWRITE    0x02  /* owner write*/
#define S_IEXEC     0x04  /* owner execute */
#define S_IOREAD    0x08  /* others read */
#define S_IOWRITE   0x10  /* others write */
#define S_IOEXEC    0x20  /* others execute */
#define S_ISUID     0x40  /* set user id for execute*/
```

Only the owner or the system manager may change the permissions of a file.

ERRORS

A successful call returns 0, -1 is returned if the caller is not entitled to change permissions or 'fname' cannot be found.

SEE ALSO

UniFLEX command perms

chown
change the ownership of a file

USAGE

```
chown(fname,ownerid)
int ownerid;
char *fname;
```

ASSEMBLER EQUIVALENT

```
sys chown, fname, ownerid
```

DESCRIPTION

This call is available only to the system manager. 'fname' is a pointer to a file name and 'ownerid' is the new user-id.

ERRORS

'0' is returned from a successful call, '-1' otherwise.

SEE ALSO

UniFLEX command owner

close
close a file

USAGE

```
close(fd)
int fd;
```

ASSEMBLER EQUIVALENT

```
<file descriptor in D>
sys close
```

DESCRIPTION

Close takes a file descriptor ('fd') as returned from system calls 'open()', 'creat(), 'dup()' or 'pipe()' and closes the associated file.

Termination of a task always closes all open files automatics1ly, but it is necessary to close files where multiple files are opened by the task and it is desired to re-use file descriptors to avoid going over the system file descriptor limit.

SEE ALSO

creat(), open(), pipe(), dup().

creat
create a new file

USACE

```
include <modes.h>
creat(fname, perm)
int perm;     1
char *fname;
```

ASSEMBLER EQUlVALENT

```
sys create, fname, perm
<file descriptor in D>
```

DESCRIPTION

Creat returns a file descriptor to a new file available for writing giving it the permissions specified in 'perm' and making the task user the owner. If, however, 'fname' is the name of an existing file it is truncated to zero length and the ownership and permissions remain unchanged.

It is unnecessary to specify writing permissions in 'perm' in order to write to the file in the current task.

The permissions allowed are defined in the include file as follows

```
/*permissions*/
#define S_IPRM      0xff  /* mask for permission bits*/
#define S_IREAD     0x01  /* owner read */
#define S_IWRITE    0x02  /* owner write*/
#define S_IEXEC     0x04  /* owner execute */
#define S_IOREAD    0x08  /* others read */
#define S_IOWRITE   0xlO  /* others write */
#define S_IOEXEC    0x20  /* others execute */
#define S_ISUID     0x40  /* set user id for execute*/
```

ERRORS

This call returns -1 if there are too many files open, the pathname cannot be searched, permission to write is denied or the file exists and is a directory.

SEE ALSO

write(), close(), chrnod(), umask()

dup, dup2
duplicate an open file descriptor

USAGE

```
dup(fd)
int fd;

dup2(fdl, fd2)
int fdl, fd2;
```

ASSEMBLER EQUIVALENT

```
<file descriptor in D>
<new file descriptor in X>
sys dup
<file descriptor in D>
```

The dup2 entry is implemented by adding 0100 to the file descriptors.

DESCRIPTION

Dup takes the file descriptor 'fd' as returned from 'open()', creat()' or 'pipe ()' and returns a new file descriptor associated with the same file.

In dup2, file descriptor 'fdl' refers to an open file and 'fd2' is a non-negative integer less than the maximum value allowed for file descriptors. Dup2 causes 'fd2' to refer to the same file as 'fdl'. If 'fd2' already referred to an open file, it is closed first.

ERRORS

A -1 is returned if there are too many files open or the given file descriptor is invalid•.

SEE ALSO

open(1), creat(), close(), pipe()

execl, execv, execlp, execvp
execute a file as a task

USAGE

```
execl(fname, arg0, arg1, …, argn,0)
char *name, *arg0, *arg1, …, *argn;

execv(fname, arglst)
char *name, *arglist[];

execlp(fname, arg0, arg1, …, argn, 0)
char *name, *argO, *argl, …, *argn;

execvp(fname, arglst)
char *name, *arglst[];
```

ASSEMBLER EQUIVALENT

```
sys exec, fname, arglst
```

The assembler call is equivalent to execv. Execl is implemented by setting up the parameters for this call.

DESCRIPTION

For the details of the operation of 'exec' see the UniFLEX manual under System Calls.

From C two forms are available. Execv is useful when the parameters are dependent on the circumstances and execl when they are known at compile time.

When a C program is executed, the argument count and arguments are made available as follows:

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

where 'argc' is the number of arguments and 'argv' points to the list of arguments. ·By convention, the first argument (argv[O]) points to the name of the running program. This is, of course, set up by the program 'exec-ing', e.g. the shell.

Execlp and execvp duplicate the shell's actions in searching for an executable file in a list of directories.

ERRORS

A return to the caller indicates an error.

SEE ALSO

fork()

exit, _exit
task termination

USAGE

```
exit(status)
int status;

_exit(status)
int status;
```

ASSEMBLER EQUIVALENT

```
<status in D> I
sys term
```

DESCRIPTION

Exit 1s the normal means of terminating a task. Exit does any cleaning up operations required before terminating such as flushing out any file buffers, but _exit does not.

A task finishing normally, that is returning from 'main', is equivalent to a call - 'exit(0)'.

The status passed to exit is available to the parent task if it is executing a wait. The status passed to _exit is available to the parent task if it is executing a 'wait.

SEE ALSO

wait()

fork
create a new task

USAGE

```
fork()
```

ASSEMBLER EQUIVALENT

```
sys fork
<new task returns here>
<old task here (pc+ 2), new task id in D>
```

DESCRIPTION

Fork creates an exact copy of the current task and continues execution at the same place in each. Each task may find out whether it is the parent or the child from the return value of the fork: the return value is 0 in the child task and the task.id of the child in the parent. This task id may be used by 'wait'.

The child copy contains the same program, data and open files as the parent but after the fork the data copies are quite distinct. If the program is not a shared-text program so is the program executable code.

Fork is the only way to start up a new task and is typically the prelude to a call to 'exec,' or 'execl'.

ERRORS

This call is unsuccessful and returns -1 if too many tasks have been created or if the system task-table is full.

SEE ALSO

exec(), wait()

getpid
get the task id

USAGE

```
getpid ()
```

ASSEMBLER EQUIVALENT

```
sys gtid
<task id in D>
```

DESCRIPTION

A number unique to the current running task is often useful in creating names for temporary files. This call returns the task's system id (as returned to its parent by 'fork').

SEE ALSO

fork()
Standard Library function mktemp

getuid, geteuid
return user id

USAGE

```
getuid ()

geteuid ()    I
```

ASSEMBLER EQUIVALENT

```
sys guid
<actual user id in D>
<effective user id in X>
```

DESCRIPTION

Getuid returns the real user id of the current task (as maintained in the password file) and geteuid returns the effective user id.

The effective user id may not be the same as the real id, as the task that is running may have the 'set id bit' set in its file giving the user of the program the permissions of the owner of that file.

SEE ALSO

setuid ()

gtty, stty
control terminal

USAGE

```
#include <sgtty.h>

gtty(fd, ttbuf)
int fd;
struct sgttyb *ttbuf;

stty(fd, ttbuf)
int fd;
struct sgttyb *ttbuf;
```

ASSEMBLER EQUIVALENT:

```
<file  descriptor in D>
sys ttyget, ttbuf

<file  descriptor' in  D>
sys ttyset, ttbuf
```

DESCRIPTION

Gtty obtains information about a terminal. Stty allows the setting of terminal parameters.

'Fd' should be a file descriptor for an open special file associated with a terminal.

The include file defines the structure returned by gtty (or assumed by stty) as follows:

```
struct sgttyb {     /* structure for 'stty' and'gtty' */
  char sg_flag,     /* mode flag - see below */
  sg_delay,         /* delay type - see below* /
  sg_kill,          /* line cancel character - default cntrl X */
  sg_erase;         /* backspace/rubout char.- default cntrl B */
  sg_speed,         /* terminal speed - unused */
  sg_spare;
};
```

The modes for sg_flag are as follows:

```
/* terminal modes */.
#define  RAW          01  /* raw - single char no mapping no echo*/
#define  ECHO         02  /* input character echo */
#define  XTABS        04  /* expand tabs on output */
#define  LCASE       010  /* map uppercase to lower case on input */
#define  CRMOD       020  /* output cr-lf for cr */
#define  SCOPE       040  /* echo backspace echo char */
#define  CBREAK     0100  /* single char i/o */
#define  CNTRL      0200  /* ignore control characters */
```

The delays for sg_delay are :

```
        /* delay  types */
        #define   DELNL      03   /* new line */
        #define   DELCR     014   /* carriage-return */
        #define   DELTB     020   /* tab */
        #define   DELVT     040   /* vertical tab */
        #define   DELFF     040   /* form-feed (as DELVT) */
```

kill
send an interrupt to a task

USAGE

```
#include <signal.h>

kill(tid, interrupt)
int tid, interrupt;
```

ASSEMBLER EQUIVALENT

```
<task id in D>
sys spint, interrupt
```

DESCRIPTION

kill sends the interrupt typt 'interrupt' to the task with id 'tid'. Both tasks, sender and receiver, must have the same effective user id unless the user is the system manager. A task id of 0 indicates that the interrupt is to be sent to all tasks associated with the user's terminal. The system manager may use a task id of -1 which will cause the interrupt to all tasks in the system except the scheduler (task 0) and the initializer (task 1). The include file contains definitions of the interrupts as follows:

```
/* UniFLEX interrupts */
#define SIGHUP       1    /* hangup */
#define SIGINT       2    /* keyboard */
#define SIGQUIT      3    /* quit */
#define SIGEMT       4    /* EMT (swi) */
#define SIGKILL      5    /* task kill can't catch/ignore */
#define SIGPIPE      6    /* broken pipe */
#define SIGSYS       7    /* bad system call argument */
#define SIGTRAP      8    /* trace */
#define SIGALRM     10    /* alarm */
#define SIGTERM     11    /* task terminate */
/*                  12       un-assigned (free for use) */
```

ERRORS

Kill returns 0 from a successful call and -1 if the task does not exist or the effective user ids do not match and the user is not the system manager.

SEE ALSO

signal()
UniFLEX command int

link
create a duplicate directory entry

USAGE

```
link(fnamel, fname2)
char *fnamel, *fname•2;
```

ASSEMBLER EQUIVALENT

```
sys link, fname1, fnam2e
```

DESCRIPTION

This call creates a new directory entry for the file 'fname1' giving it the name 'fname2'. The contents of the file and its attributes and permissions are unchanged.

A link may be made from any directory to any other provided they are both on the same disc drive.

ERRORS

A successful call returns '0', the '-1' error return is made if:

```
fname1 does not exist,
fname2 already exists,
wrice permission in the directory of fname2 is denied,
fnamel is a directory,
the directories are on different devices,
the file already has too many links.
```

SEE ALSO

```
UniFLEX command link
unlink()
```

lock
lock a task in memory

USAGE

```
lock(flag)
```

ASSEMBLY EQUIVALENT

```
sys lock, flag
```

DESCRIPTION

Lock will prevent the calling task from being swapped out to disc. This call is only available to the system manager.

A value for 'flag' non-zero causes the task to be locked, zero to be unlocked.

ERRORS

A -1 1s returned if the user is not the system manager.

lrec, urec
lock or unlock a file record

USAGE

```
lrec(fd, count)
int fd, count;

urec(fd)
int fd;
```

ASSEMBLER EQUIVLENT

```
<file descriptor in D>
sys lrec, count

<file descriptor in D>
syn urec
```

DESCRIPTION

Details of the effects of these calls should be sought in the UniFLEX manual under 'lrec' and 'urec'.

ERRORS

Zero is returned from a successful call. A -1 1s returned if the 'fd' is invalid, the file is a special file, a pipe or a directory, record is locked by another active task or the lock table is full.

lseek
position 1n file

USAGE

```
long lseek(fd, position, type)
int fd, type;
long position;
```

ASSEMBLER EQUIVALENT

```
<file descriptor in D>
sys seek, positionhi, positionlo, type
<hi position in X>
<lo position 1n D>
```

DESCRIPTION

The read or write pointer for the open file with file descriptor 'fd' is positioned by lseek to the specified place in the file. The 'type' indicates from where 'position' is to be measured:

```
if 0,    from the beginning of the file
if 1,    from the current location or
if 2,    from the end of the file.
```

Seeking to a location beyond the end of a file open for writing and then writing to it creates a 'hole' in the file which appears to be filled with zeros from the previous end to the position sought.

The returned value is the resulting position in the file unless there is an error, so to find out the current position use 'lseek(fd,01,1);'.

WARNINGS

The argument 'position' MUST be a long integer. Constants should be explicitly made long by appending an 'l', as above and other types should be converted using a cast. e.g. 'lseek(fd,(long)pos,l);'.

ERRORS

A -1 is returned if 'fd' is a bad file descriptor, an attempt to seek on a pipe or to a position before the beginning of a· file.

SEE ALSO

open(), creat();
Standard Library function fseek·

mknod
create a special file or directory

USAGE

```
#include <modes.h>

mknod(fname,desc, address)
char *fname;
```

ASSEMBLER EQUIVALENT

```
sys crtsd, fname, desc, address
```

DESCRIPTION

This call may be used by the system manager to make a new special file (device) or directory. 'Fname' should point to a string containing the desired name of the file. 'Desc' is a descriptor specifying the desired mode (file type) and permissions of the new file. 'Address' is a 16 bit internal device number but for directories should be zero.

The include file defines the possible values for 'desc' as follows:

```
/* file modes */
#define S_IFMT      0xff00      /* mask for type of file */
#define S_IFREG     0x0100      /* normal file */
#define S_IFBLK     0x0200      /* block special*/
#define S_IFCHR     0x0400      /* character special */
#define S_IFDIR     0x0800      /* directory */

/* permissions */
#define S_IPRM      0xff  /* mask for permission bits*/
#define S_IREAD     0x01  /* owner read */
#define S_IWRITE    0x02  /* owner write*/
#define S_IEXEC     0x04  /* owner execute */
#define S_IOREAD    0x08  /* others read */
#define S_IOWRITE   0x10  /* others write */
#define S_IOEXEC    0x20  /* others execute */
#define S_ISUID     0x40  /* set user id for execute*/
```

ERRORS

Zero is returned if the file has been successfully made, -1 if the caller is not the system manager or if the file already exists.

SEE ALSO

UniFLEX commands crdir and makdev

PROGRAMMING NOTES

When using mknod to create directory files, you should use link to generate the "." and ".." entries in the newly created directory. If this is not done, the UniFLEX diagnostic routines "blockcheck" and "fdncheck" will report file structure errors. By convention, the first entry in a directory is named "." and is a link to the directory itself. The second entry in a directory is named ".." and is a link to the parent of the directory.

mount, umount
mount or unmount a file system

USAGE

```
mount(sname, fname, mode)
int mode;
char *sname, *fname;

umount(sname)
char *sname;
```

ASSEMBLER EQUIVALENT

```
sys mount, sname, fname, mode

sys unmnt, sname
```

DESCRIPTION

Mount may be used to 'connect' a device (special file) to a branch of the file system tree of directories. 'Sname' should point to a string representing the pathname of an existing special file. 'Fname' should point to the pathname of an existing directory which is to become the root directory of the file system on the device. The previous contents of this directory is 'hidden' for the duration of the mount. 'Mode'. should be zero where read and write access is required and non-zero for read-only access.

Umount reverses the effect of 'mount'.

ERRORS

Both calls return 0 if successful. Mount returns -1 if 'sname' does not exist, is not a special file or is already mounted, if 'fname' does not exist, is un-writable or is already in use or if too many devices are mounted. Umount returns -1 if the file system has active filee or is not mounted.

SEE ALSO

UniFLEX commands /etc/mount and /etc/unmount

nice
set program priority

USAGE

```
nice(priority)
int priority;
```

ASSEMBLER EQUIVALENT

```
<priority in D>
sys nice
```

DESCRIPTION

The scheduling priority of the process is augmented by 'priority'. Positive priorities get less service than negative priorities.

Negative increments are ignored except on behalf of the super-user. The priority range is 'limited to the range -20 (most urgent) to +20 (least urgent).

The priority process is passed to a child process by fork(). For a privileged process to return to normal priority from an unknown state, nice should be called successively with arguments -40 (goes to -20 because of truncation), 20 (to get 0), then O (to maintain compatibility with previous versions of this call).

open
open a file for read/write access

USAGE

```
open(fname, mode)
char *fname;
```

ASSEMBLER EQUIVALENT

```
sys open, fname, mode
<file descriptor in D>
```

DESCRIPTION

This call opens an existing file for reading if 'mode' is 0, writing if 'mode' is 1 or reading and writing if 'mode' is 2, 'Fname' should point to a string representing the pathname of the file. Open returns an integer as 'file descriptor' which should be used by i/o system calls referring to the file.

The position where reads or writes start is at the beginning of the file.

ERRORS

A '-1' is returned if the file does not exist, the pathname cannot be searched, too many files are already open or if the file permissions deny the requested mode.

SEE ALSO

creat(), read(), write(), dup(), close()

pause
halt and wait for interrupt

USAGE

```
pause()
```

ASSEMBLER EQUIVALENT

```
sys  stop
```

DESCRIPTION

Pause may be used to halt a task until an interrupt is received from 'kill' or 'alarm'. Pause always returns -1.

SEE ALSO

kill{), alarm{), signal{)
UniFLEX command int

pipe
create a pipe between tasks

USAGE

```
pipe(fd)
int fd[2];
```

ASSEMBLER EQUIVALENT

```
sys crpipe
<read file descriptor in D>
<write file descriptor in X>
```

DESCRIPTION

For a full description of the operation of pipes, see the UniFLEX manual under 'crpipe'.

Pipe should be passed the address of a two-integer array into which the returned file descriptors will be placed. The one with the lower address is the read file descriptor, the other is the write file descriptor.

ERRORS

'0' is returned from a successful call, -1 if too many files are already open. An interrupt (SIGPIPE) is generated if a write is attempted on a pipe where the other end is closed.

SEE ALSO

The UniFLEX shell
read(), write(), fork()

profil
profile a running task

USAGE

```
profil (buffer, bsize, prpc, scale)
char *buffer;
int bsize, prpc, scale;
```

ASSEMBLER EQUIVALENT

```
sys profil, prpc, buffer, bsize, scale
```

NOTE that the order of the arguments differs in the C call and the assembler call for portability reasons.

DESCRIPTION

The details of this call should be sought in the UniFLEX manual under 'profile'. Users familiar with profil on other systems should note the following:

'bsize' is the size of the buffer in WORDS not bytes,
'scale' must be a power of two <= 128,
the 'grain' of the profile is 1/10th of a second.

ERRORS

No error is returned by profil.

SEE ALSO

Standard Library function monitor

read
read from a file

USAGE

```
read(fd, buffer, count)
int fd, count;
char *buffer;
```

ASSEMBLER EQUIVALNT

```
<file descriptor in D>
sys read, buffer, count
<bytes read 1n D>
```

DESCRIPTION

The file descriptor 'fd' is an integer which should have been returned by a successful call to 'open', 'creat','dup' or 'pipe'. 'Buffer' is a pointer to space with at least 'count' bytes of memory into which read will put the data from the file.

It is guaranteed that at most 'count' bytes will be read but often less will be, either because the file represents a terminal and input stops at the end of a line, or end-of-file has been reached.

ERRORS

Read returns the number of bytes actually read (0 at end-of-file) or -1 for physical i/o errors, a bad file descriptor or a ridiculous 'count'.

SEE ALSO

open(), creat(), dup(), pipe()

setstack, stacksize
set or obtain stack reservation

USAGE

```
setstack(size)
int size;

stacksize()
```

DESCRIPTION

A block of memory near the top of the address space is reserved by the system on execution of a task. The size of this space depends on the number of bytes occupied by the arguments of the 'exec' call which set up the task. If memory is required for the stack below this reserved space, it must be requested from the system; in assembler by means of a call to 'stack'.

The run-time support of a program generated by the C compiler normally ensures that the task always has at least 128 bytes of reserved stack space and does the necessary housekeeping, so it is normally unnecessary for the programmer to worry about the stack.

However, cc has an option (+S) which allows the user to specify that this stack-checking code be omitted from the program. It will make a small difference in code size and running time and should only be used on extremely time critical code.

If a program has been compiled with this option, it will be necessary to reserve stack space to avoid a possible attempt to access memory in the 'no-go' area between the top of the data area and the reserved stack space. Setstack requests at least 'size' bytes, to be available.

In order to determine how much to reserve, the prcgram may be compiled in the normal way without the +S option and with a call to stacksize near the end of execution. Stacksize returns the current size of the reserved stack space measured from the position of the stack pointer at the start of execution. Note that programs with recursive functions whose depth of recursion depends on input data will not provide a reliable indication via stacksize.

Setstack called with a value·of 'size' greater than the amount of available address space will cause program termination with a message on the terminal and a core dump.

setuid
set user id

USAGE

```
setuid(uid)
```

ASSEMBLER EQUIVALENT

```
<user id in   D>
sys suid
```

DESCRIPTION

This call may be used to set the actual and effective ids for the current task. Setuid only works if the caller is the system manager or 'uid' matches the actual user id of the caller.

ERRORS

'0' is returned from a successful call, '-1' otherwise.

SEE ALSO

getuid()

signal
catch or ignore interrupts

USAGE

```
#include <signal.h>
(*signal(interrupt, address))()
(*address)();
```

(Which means: 'signal' returns a pointer to a function, 'address' is a pointer to a function.)

ASSEMBLER EQUIVALENT

```
sys cpint, interrupt, address
<old address in D>
```

DESCRIPTION

Full details of this call should be sought in the UniFLEX manual under 'cpint'. The interrupts used by UniFLEX are defined in the header file as follows:

```
/* UniFLEX interrupts */
#define SIGHUP       1    /* hangup */
#define SIGINT       2    /* keyboard */
#define SIGQUIT      3    /* quit */
#define SIGEMT       4    /* EMT (swi) */
#define SIGKILL      5    /* task kill can't catch/ignore */
#define SIGPIPE      6    /* broken pipe */
#define SIGSYS       7    /* bad system call argument */
#define SIGTRAP      8    /* trace */
#define SIGALRM     10    /* alarm */
#define SIGTERM     11    /* task terminate */
/*                  12       un-assigned (free for use) */

/* special addresses */
#define SIG_DFL      0    /* default */
#define SIG_IGN      1    /* ignore */
```

ERRORS

A '-1' is returned if the interrupt is less than 1 or more than 12.

SEE ALSO

UniFLEX command int
kill()

stat, fstat
get file status

USAGE

```
#include <stat.h>
#include <modes.h>

stat(fname, buffer)
char *fname;
struct stat *buffer;

fstat(fd, buffer)
int fd;
struct stat *buffer;
```

ASSEMBLER EQUIVALENT

```
sys status, fname, buffer

<file descriptor in D>
sys ofstat, buffer
```

DESCRIPTION

These calls obtain details about the status of a file. Stat will refer to any file by then name 'fname', fstat to an open file by its file descriptor 'fd'.

Each call will fill the memory pointed to by 'buffer' with a structure declared in the first include file as:

```
struct stat   /* structure returned by stat*/
{
  int         st_dev ;    /* device number */
  int         st_ino ;    /* fdn number */
  unsigned    st_mode ;   /* file mode and permissions */
  char        st_nlink ;  /* file link count */
  int         st_uid ;    /* file owner's user id */
  long        st_size ;   /* file size in bytes */
  long        st_mtime ;  /* last modified time */
  long        st_spr ;    /* spare - future use only */
};
```

The second include file enables the interpretation of 'st_mode' as follows:

```
/* file modes */

#define S_IFMT      0xff00      /* mask for type of file */
#define S_IFREG     0x0100      /* normal file */
#define S_IFBLK     0x0200      /* block special*/
#define S_IFCHR     0x0400      /* character special */
#define S_IFDIR     0x0800      /* directory */
```

```
/*permissions*/

#define S_IPRM      0xff  /* mask for permission bits*/
#define S_IREAD     0x01  /* owner read */
#define S_IWRITE    0x02  /* owner write*/
#define S_IEXEC     0x04  /* owner execute */
#define S_IOREAD    0x08  /* others read */
#define S_IOWRITE   0x10  /* others write */
#define S_IOEXEC    0x20  /* others execute */
#define S_ISUID     0x40  /* set user id for execute*/
```

ERRORS

'0' is returned from a successful call, '-1' if the status is unavailable.

SEE ALSO

UniFLEX command dir (with the '-1' option)

stime
set the system time

USAGE

```
stime(tp)
long *tp;
```

ASSEMBLER EQUIVALENT

```
<timehi in X>
<timelo in D>
sys stime
```

DESCRIPTION

The system manager may set the system's idea of the time snd date with this call. The time is measured in seconds from 0000 January 1, 1980.

WARNINGS

The desired time is passed by a pointer to the actual desired time, NOT by value.

ERRORS

A '-1' is returned if the caller is not the system manager, '0' otherwise.

SEE ALSO

UniFLEX command date
Standard Library function ctime
time()

sync
update file system

USAGE

sync()

ASSEMBLER EQUIVALENT

sys update

DESCRIPTION

This call is used to cause any data destined for disc to be written out.

ERRORS

'0' is always returned

time, ftime
get current time and date

USAGE

```
long time(0)

long time(tp}
long *tp;

#include <timeb.h>
ftime(tbuf)
struct timeb *tbuf;
```

ASSEMBLER EQUIVALENT

```
sys time, tbuf
```

NOTE: This is equivalent to 'ftime' only.

DESCRIPTION

Time returns the long integer value of the system date and time and, if its argument is non-zero, places this value in the long integer pointed to by 'tp'.

Ftime gives more information by filling in a structure defined in the header file1as follows:

```
/* 'ftime' structure */
struct timeb {
  long  time ;      /* time in seconds */
  char  tm_tik ;    /* ticks in second (tenths) */
  int   timezone ;  /* time zone */
  char  dstflag ;   /* daylight savings flag */
};
```

'Timezone' is the number of minutes west of Greenwich.

SEE ALSO

UniFLEX command date
Stadard Library function ctime
stime()

times
get task times

USAGE

```
#include <times.h>

times(buffer)
struct tbuffer *buffer;
```

ASSEMBLER EQUIVALENT

sys ttime, buffe1

DESCRIPTION

Times may be used to obtain time-accounting information about the cuf-rent task and its child tasks.

The times are placed in the structure at 'buffer' and appear as in the include file as follows:

```
/* 'times' structure */

struct tbuffer {
   char  ti_usr[3] ; /* task's user time*/
   char  ti_sys[3] ; /* task's system time*/
   long  ti_chu ;    /* children's user time*/
   long ti-chs ;     /* children's system time*/
};
```

All times are in tenths of seconds. Ti_usr and ti_sys may be converted to long integers using '13tol' in the Standard Library.

ERRORS

No error is issued.

SEE ALSO

time()

umask
set default permissions

USAGE

```
#include <modes.h>

umask(perm)
int perm;
```

ASSEMBLER EQUIVALENT

```
sys defacc, perm
```

DESCRIPTION

At each call of 'creat' or 'mknod' the permissions specified in the call are modified by a set of default permissions maintained for the task. The requested permissions are logically anded with the complement of this default mask. The effect of this is to restrict permissions to those not included in the mask.

Umask sets the default mask and returns the previous value of the mask. Unless modified by umask, the default mask is 0, i.e. no restrictions. Child tasks inherit the default mask. The value of 'perm' should be a combination of the allowable permission codes as defined in the include file as follows:

```
/* permissions */
#define S_IPRM      0xff  /* mask for permission bits*/
#define S_IREAD     0x01  /* owner read */
#define S_IWRITE    0x02  /* owner write*/
#define S_IEXEC     0x04  /* owner execute */
#define S_IOREAD    0x08  /* others read */
#define S_IOWRITE   0x10  /* others write */
#define S_IOEXEC    0x20  /* others execute */
#define S_ISUID     0x40  /* set user id for execute*/
```

SEE ALSO

creat(), mknod(), chmod()

unlink
remove directory entry

USAGE

unlink(fname)

ASSEMBLER EQUIVALENT

sys unlink, fname

DESCRIPTION

Unlink deletes the directory entry whose name is pointed to by 'fname'. If the entry was the last link to the file the file itself is deleted and the disc space occupied made available for re-use. If the file is open in any active task, the deletion of the actual file is delayed until the file is closed.

ERRORS·

'0' is returned from a successful call, '-1' if the file does not exist, its directory is write-protected or cannot be searched, the file is a directory or if the file is an executable program currently running.

SEE ALSO

UniFLEX command kill
link()

wait
wait for task termination

USAGE

```
wait(status)
int *status;
```

ASSEMBLER EQUIVALENT

```
sys wait
<task id in D>
<term status in X>
```

DESCRIPTION

Wait is used to halt the current task until a child task has terminated. The call returns the task id of the terminating task and places the status of that task in the integer pointed to by 'status'. A wait must be executed for each child task spawned;

The high (most significant) byte of the status is the low byte of the argument of the 'exit' or '_exit' call in the child task. A normally terminating C program with no call to 'exit' or '_exit' has an implied call of '_exit(0)'.

The low byte of the status is normally zero, but if not it indicates a program interrupt of some kind; see 'signal' for a list of possible values. In addition, if the most significant bit of the low byte is set, a core dump has been made.

WARNINGS

For portabitity reasons, the order of the bytes in the status word is different in the C call and the assembler call.

ERRORS

A '-1' is returned if there is no child to be waited for.

SEE ALSO

fork(), signal(), exit(), _exit()

write
write to a file

USAGE

```
write(fd, buffer, count)
int fd, count;
char *buffer;
```

ASSEMBLER EQUIVALENT

```
<file descriptor in D>
sys wait, buffer, count
```

DESCRIPTION

'Fd' must be a value returned by 'open', 'creat', 'dup' or 'pipe'. 'Buffer' should point to an area of memory from which 'count' bytes are to be written. Write returns the actual number of bytes written and if this is different from 'count' an error has occurred.

Writes in multiples of 512 bytes to file offset boundaries of 512 bytes are the most efficient.

ERRORS

A '-1' is returned if 'fd' is a bad file descriptor, if 'count' is ridiculous or on a physical i/o error.

SEE ALSO

creat(), open(), pipe()

SYSTEM CALL CONVERSION

The names of system calls available to C programmers are designed to allow existing C programs to compile and run without modification. Users already familiar with UniFLEX system calls may use the cross reference below to find the C call equivalent.

UniFLEX call	C call equivalent	Description
alarm	alarm	cause delayed interrupt
break	brk	extend memory address
cdata	cdata	request contiguous data
chacc	access	check access permission
chdir	chdir	change directory
chown	chown	change file owner
chprm	chmod	change access perm
close	close	close file
cpint	signal	catch program interrupt
create	creat	create a file
crpipe	pipe	create a pipe
crtsd	mknod	make special or directory file
defacc	umask	set default access
dup	dup	duplicate open file
dups	dup2	duplicate specific file
exec	execv	exececute program
fork	fork	fork a new task
gtid	getpid	get task id
guid	getuid,geteuid	get user id
ind	(not applicable)	indirect call
indx	(not applicable)	index indirect call
link	link	link to file
lock	lock	lock task in memory
lrec	lrec	lock file record
mount	mount	mount device
ofstat	fstat	get open file status
open	open	open file
profil	profil	profile taslc
read	read	read from file
seek	lseek	seek to file position
setpr	nice	set priority bias
spint	kill	send program interrupt
stack	(see setstack in C calls)	grow stack
status	stat	get file status
stime	stime	set time
stop	pause	stop til interrupt
suid	setuid	set user id
term	_exit	terminate task
time	ftime	get time
trap	(not supported)	set swi2 trap vector
ttyget	gtty	get tty status
ttyset	stty	set tty status
unlink	unlink	remove file link
unmnt	umount	unmount device
update	sync	update file systems
urec	urec	unlock file record
wait	wait	wait for task termination
write	write	write to file

Interfacing to Assembly Code

In the normal course of events, it is entirely unnecessary to substsitute assembly code for C code in a program. The small savings in object code size and execution time are not worth the disadvantages of using a lower-level language. However, for the rare occasions when it is essential to use assembly code the following describes how C functions are structured.

All code produced by the C compiler is designed to allow the assembler to produce relocatable object modules. This means that there are no 'org' directives, all code is in the 'text' segment, initialized data such as strings are in the 'data' segment and unitialised data is in the 'bss' segment.

On entry to a function, each argument occupies as many bytes as is necessary for its type. The layout of the stack on entry to a function called in the form

```
foo(arg0,argl,arg2,… argn)
```

is as follows:

```
                argn
                …
                arg2
                argl
                arg0
stack pointer-> return address
```

Immediately on entry the U and Y registers are pushed onto the stack so as to preserve their values over the function call. The first argument is the six bytes past the S register (e.g. 6,s) This results in the stack looking like:

```
                argn
                …
                arg2
                argl
                arg0
          return address
             U register
stack pointer-> Y register
```

Space on the stack for local variables, if required is then reserved below this by moving the stack pointer down.

The returned value from a function is passed back in a different form depending on the type of the returned value. For all types other than long, float or double, the value is returned in the D register. For the others the value is in a fixed location in the bss segment where eight bytes are reserved which has the label.'_flacc'. In addition, the X register points to '_flacc' and hence the returned value.

Note that if a function returns a value type char, it will be converted to int; this means that the actual byte is in the B register and this is sign-extended into the A register.

The calling sequence, then, need not worry about the U and Y registers as a C function will preserve their values but the stack must be restored as the C function will not restore it. All that is required is to push the arguments onto the stack in REVERSE ORDER, call the function using a 'jsr' instruction and then restore the stack by the sum of the bytes pushed before the call.

In order to write a function which behaves like a C function, only three rules have to be observed:

1. the values of the U and Y registers must be preserved,
2. the code must allow a relocatable module to be produced,
3. the stack pointer must be preserved.

The latter should probably be implemented by restoring to the state on entry and executing an 'rts' instruction.

If the assembly code function is to return a value then it should imitate a C function as above.

Assembly-code functions should be kept in separate files and not written in using the '#asm' compiler directive. In order that the loader can link the function to the rest of the program, the name of the function should be declared 'global'. Note that a relocatable module must have a name; see the documentation of 'relasmb'.

Index