qsort

quick sort

SYNOPSIS

#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size, int (*compare)(const void *, const void *))

DESCRIPTION

The **qsort** function is an implementation of the "quick sort" algorithm. It sorts an array of **nmemb** objects, the first object of which **base** points to. Each object is **size** bytes in size. The array is sorted in ascending order using a programmer supplied comparison function (pointed to by **compare**).

The **compare** function takes two arguments which point to two objects to be compared. The **compare** function should return a positive value if the first object is considered to be "greater-than" the second object; a value of zero if the objects are considered to be "equal"; otherwise a negative value.

The order "equal" elements have in the resulting sorted array is undefined (in sort terminology, **qsort** is "unstable").

Two other sort routines are provided with the same calling sequence as **qsort**, for use where the data set matches special circumstances. See the "isort" and "ssort" descriptions for more information.

RETURN VALUE

None

RELATED FUNCTIONS

isort, ssort

rand

integral random number

SYNOPSIS

#include <stdlib.h>

int rand(void)

DESCRIPTION

The **rand** function returns a sequence of pseudo-random integers in the range of zero to 2**31. No other library function calls **rand**; but the "frand" function shares the same "seed". The "seed" may be set for **rand** using the "srand" function.

RETURN VALUE

See above

RELATED FUNCTIONS

frand, srand

read

low level file read

SYNOPSIS

```
#include <stdio.h>

int read(int handle, char *buf, size_t n)
```

DESCRIPTION

The **read** function reads **n** bytes, from the file opened with **handle**, to the buffer pointed to by **buf**.

The file must have been previously opened by a call to "open" or "creat", and be readable.

As this function is not supported in the ANSI standard library, its use in new work is discouraged (see the "fopen", "fclose", "fread", and "fwrite" functions for the recommended replacements).

RETURN VALUE

The **read** function returns the number of bytes actually read. A return value of zero indicates EOF, (-1) indicates an error ("errno" will be set appropriately).

If the file being read has been opened in a "text" mode, and the host OS performs CR/LF mappings, the return value may be less than what was requested WITHOUT indicating that no more bytes are available in the file (beyond those fetched by the current **read** call).

Note the difficulty that reading a 65535 byte file on a computer with a 16 bit "int" size causes (the non-error and error return values are the same).

RELATED FUNCTIONS

close, creat, dup, dup2, open, write

realloc

change allocation of heap memory

SYNOPSIS

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

DESCRIPTION

The **realloc** function changes the size of an existing allocated region of heap memory. The contents of the region will be unchanged up to the lesser of the old and new **size** values. If the new **size** is larger, the value of the newly allocated portion of the region is undefined. If the value of **ptr** is NULL, the function behaves like the "malloc" function called with **size** as the parameter. If the requested space is unavailable, the region pointed to by **ptr** is not changed. If **size** is zero, and the value of **ptr** wasn't NULL, the **ptr** region is returned to the heap free storage pool.

If the value of **ptr** wasn't NULL, and the value wasn't the result of an earlier call to "calloc", "malloc" or "realloc", the results are undefined.

RETURN VALUE

If it is impossible to satisfy the request, or **size** is zero, a NULL pointer is returned. Otherwise, a pointer is returned to the start of the (possibly moved), allocated region.

RELATED FUNCTIONS

addfree, calloc, cfree, free, malloc

remove

delete file

SYNOPSIS

#include <stdio.h>

int remove(const char *pathname)

DESCRIPTION

The **remove** function deletes the file specified by **pathname**. If the file is currently open the results are undefined.

RETURN VALUE

If the deletion is successful, zero is returned; otherwise non-zero.

RELATED FUNCTIONS

rename, unlink

rename

change file name

SYNOPSIS

```
#include <stdio.h>

int rename(const char *oldname, const char *newname)
```

DESCRIPTION

The **rename** function changes the name of the file pointed to by **oldname**, into the name pointed to by **newname**. If a file already exists using the **newname** string as a name, the results are undefined.

RETURN VALUE

If the renaming is successful, zero is returned; otherwise non-zero. If not successful, the original name is still in force.

RELATED FUNCTIONS

None

restorefp

restore floating point pseudo registers

SYNOPSIS

```
#include <conc.h>

void restorefp(FPstate *fp)
```

DESCRIPTION

The **restorefp** function is only required for programs which use floating point, and are running on Transputers which lack hardware floating point support. It allows restoring the floating point emulation registers which were saved as a result of a previous call to "savefp". As mentioned in the "savefp" description, use of these functions is only required when floating point operations are being done by processes at both low and high priority.

See the "savefp" function description for more information about how these functions should be used.

RETURN VALUE

None

RELATED FUNCTIONS

savefp

rewind

rewind stream

SYNOPSIS

```
#include <stdio.h>

void rewind(FILE *stream)
```

DESCRIPTION

The **rewind** function rewinds the current read/write position for **stream** to the very beginning. It also clears the "error" and "eof" flags for the **stream**.

RETURN VALUE

None

RELATED FUNCTIONS

fseek, ftell, lseek

rindex

reverse search string for character

SYNOPSIS

#include <string.h>

char *rindex(const char *ptr, int c)

DESCRIPTION

The **rindex** function searches a string, pointed to by **ptr**, for the last occurrence of the character **c**. A pointer to where the character was found is returned, or NULL, if a '\0' string terminator was found instead. The '\0' character is considered part of the string and may be searched for. This function is called "strrchr" in ANSI libraries, and use of that name is recommended for all new work.

RETURN VALUE

See above

RELATED FUNCTIONS

index, strchr, strrchr

savefp

save floating point pseudo registers

SYNOPSIS

```
#include <conc.h>

void savefp(FPstate *fp)
```

DESCRIPTION

The **savefp** function is only required for programs which use floating point, and are running on Transputers which lack hardware floating point support. It allows saving the floating point emulation registers for later restoration with a call to "restorefp". Use of these functions is only required when floating point operations are being done by processes at both low and high priority.

On floating point Transputers, a context switch from low to high priority results in the automatic saving of the low priority floating point registers by the processor. For Transputers which lack floating point support, this action is simulated by having the high priority process call "savefp" to save the previous "pseudo floating point register" values prior to performing any floating point operations. When the process is finished with all floating point use, a matching call to "restorefp" may be used to restore the previous values.

Using this technique, the floating point emulation routines may be shared by both low and high priority processes without conflict. Note that only high priority processes which use floating point will require this protection.

The parameter to **savefp** should be a structure of type FPstate (**fp**), whose storage duration lasts between the time **savefp** is called, and the (later), matching call to "restorefp".

RETURN VALUE

None

RELATED FUNCTIONS

restorefp

scanf

formatted read

SYNOPSIS

```
#include <stdio.h>

int scanf(const char *format, ...)
```

DESCRIPTION

The **scanf** function reads from the "stdin" stream. The read is formatted under the control of the **format** argument, which specifies the legal input text sequences and conversion instructions. Subsequent arguments (if any), are used as pointers to the objects which receive converted data from the input. If there are too few arguments to match the requirements of the **format** string, the results are undefined. If arguments remain after meeting the requirements of the **format** string, they are evaluated and ignored. **scanf** returns when the input text fails to match the requirements of the **format** string, or when the **format** string is exhausted.

The **format** string consists of a '\0' terminated character sequence composed of:

1. Whitespace characters. These match zero or more whitespace characters read from the input (the whitespace read is ignored).

2. Ordinary characters (non-'%'). These must match exactly with characters read from the input (which are ignored after being read).

3. Format specifications (starting with '%'). These are generally used to describe what input text is a legal match for this part of the **format** string, how to read and convert the text, and how to store the result.

Format Specification

After the '%' character, the following fields make up a format specification:

1. An optional assignment suppression character ('*'). This causes any text read while matching this format specification to be discarded. No argument is need in the call to **scanf** to store the results of this conversion.

2. An optional decimal integer which specifies the maximum field width.

3. An optional 'h', 'l', or 'L' character, which defines the size of the receiving object. The 'd', 'i', and 'n' conversion specifiers are preceded with 'h' to indicate the corresponding argument is a pointer to "short int", or by 'l', to indicate a pointer to "long int". The 'o', 'u', and 'x' conversion specifiers are treated in the same fashion, except the parameters are taken to be "unsigned short int", and "unsigned long int", respectively. The 'l' character may be used with the 'e', 'f', and 'g' conversion specifiers to indicate a pointer to "double" ("float" is the default), or, alternatively, 'L' may be used to indicate a pointer to "long double". If the 'h', 'l', or 'L' characters are used with any other conversion specifier, the results are undefined.

4. Finally, the character which specifies the desired input sequnce to match and convert.

Note that input whitespace is ignored for the purposes of reading and matching conversions (except with the '[', 'c' and 'n' conversion specifiers).

Conversions

The following are the legal conversion characters:

'c' - Matches a sequence of input characters up to the specified field width (default of one). The corresponding argument will be a pointer to a character array large enough to store the matched sequence.

'd' - Matches an optionally signed decimal integer. The format matched is equivalent to that accepted by the "strtol" function with a "base" argument of ten. The corresponding argument will be a pointer to "int".

'e'
'E'
'f'
'g'
'G' - Matches an optionally signed floating point number. The format matched is equivalent to that accepted by the "strtod" function. The corresponding argument will be a pointer to "float".

'i' - Matches an optionally signed integer, whose format is either decimal, hexadecimal, or octal. The format matched is equivalent to that accepted by the "strtol" function with a "base" argument of zero. The corresponding argument will be a pointer to "int".

'n' - No input is matched or read. The corresponding argument will be a pointer to "int", into which a current count of the input characters read by this call to **scanf** will be written. Evaluation of this conversion specifier doesn't affect the return value from **scanf** (the # of successful assignments).

'o' - Matches an optionally signed octal integer. The format matched is equivalent to that accepted by the "strtoul" function with a "base" argument of eight. The corresponding argument will be a pointer to "unsigned int".

'p' - Matches an implementation defined sequence which is produced by the 'p' conversion specifier in the "printf" function. This "address" value takes the form of a normal hexadecimal integer in the **Transputer Toolset**.

's' - Matches a sequence of non-whitespace characters. The coresponding argument will be a pointer to a character array large enough to store the string (and the terminating '\0' which is automatically added).

'u' - Matches an optionally signed decimal integer. The format matched is equivalent to that accepted by the "strtoul" function with a "base" argument of ten. The corresponding argument will be a pointer to "unsigned int".

'x'
'X' - Matches an optionally signed hexadecimal integer. The format matched is equivalent to that accepted by the "strtoul" function with a "base" argument of sixteen. The corresponding argument will be a pointer to "unsigned int".

'[' - Matches a non-empty set of characters from a set of "expected" characters. The corresponding argument will be a pointer to a character array large enough to store the string (and the terminating '\0' which is automatically added). The conversion specifier includes all subsequent characters in the **format** string until a matching ']' is encountered. The characters between the brackets constitute the "expected" set, unless a '^' character immediately follows the '[' (in which case all characters EXCEPT THOSE listed, constitute the "expected" set). As a special case, the ']' is allowed to be in the "expected" set if it immediately follows the '[' (or to not be in the "expected" set, if it immediately follows a "[^"). When the special case is in effect, the concluding ']' will be the second one encountered in the **format** string.

'%' - Matches a literal '%' in the input. This allows "%%" to behave like an ordinary character in the **format** string.

Conversion specifications other than these produce undefined results.

Caveat

There is a maximum field width (per-conversion), of 509 characters, for this implementation.

RETURN VALUE

The **scanf** function returns EOF if an input failure occurs before any conversions. Otherwise, **scanf** returns the number of input sequences which were matched, converted, and assigned. The number of sequences returned, may be less than the number provided for in the **format** string, if a match fails, or EOF is encountered.

If EOF occurs in the middle of matching an input sequence, such that the match may already be considered successful (not counting optionally matched whitespace), the EOF is considered to merely terminate the current match. Any remaining conversions, however, will be immediately aborted if they require input.

RELATED FUNCTIONS

fscanf, sscanf, strtod, strtol, strtoul

EXAMPLES

The program fragment:

```
#include <stdio.h>
int n,i; float x; char name[49];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
24 54.31E-1 johnson
```

will assign the value 3 to 'n', 24 to 'i', 5.431 to 'x', and store the string "johnson\0" into array "name". Or:

```
#include <stdio.h>
int i; float x; char name[51];
scanf("%2d%f%*d %[0123456789]", &i, &x, name);
```

with the input line:

```
45678 0123 55:72
```

will assign the value 45 to 'i', 678.0 to 'x', will skip 0123, and store the string "55\0" into array "name". The next character to be read will be ':'.

To repeatably read a quantity, unit of measure, and the name of an item, the following code fragment:

```
#include <stdio.h>
int cnt; float quant; char units[25]; item [25];
while(!eof(stdin) && !ferror(stdin))
      {
      cnt = scanf("%f%24s of %24s",&quant, &units, &item);
      scanf("%*[^\n]"); /* Slurp rest of line */
      }
```

with input lines:

```
1 quart of milk
100 degrees Celsius
loads of fun
```

will produce a series of actions equivalent to:

```
1.    quant = 1.0; strcpy(units,"quart"); strcpy(item,"milk"); cnt = 3;
2.    quant = 100.0; strcpy(units,"degrees"); cnt = 2; /* No "of" */
3.    cnt = 0; /* No quantity */
4.    cnt = EOF; /* No input data */
```

SemAlloc/SemFree

semaphore allocation

SYNOPSIS

```
#include <conc.h>

Semaphore *SemAlloc(void)

void SemFree(Semaphore *s)
```

DESCRIPTION

The **SemAlloc** function returns a pointer to an initialized semaphore. The memory for the semaphore is obtained from the heap.

The **SemFree** function takes a semaphore, **s**, previously allocated using **SemAlloc**, and returns it to the heap free storage pool.

Note that semaphores may also be statically allocated (either globally, or as "auto" variables), if the dynamic creation and destruction allowed by **SemAlloc** and **SemFree** isn't required. To "statically" allocate/initialize a semaphore, the following might be used:

```
Semaphore xyz = SEMAPHOREINIT;
```

For additional information, see the included paper by Jeff Mock: **Processes, Channels and Semaphores**.

RETURN VALUE

See above

RELATED FUNCTIONS

HSemP, HSemV, SemP, SemV

SemP/_SemP/SemV/_SemV

mono-priority semaphores

SYNOPSIS

```
#include <conc.h>

int SemP(Semaphore s)

int _SemP(Semaphore *s)

int SemV(Semaphore s)

int _SemV(Semaphore *s)
```

DESCRIPTION

The **SemP** and **SemV** macros implement semaphore operations on the Transputer. These macros must not be used in situations where the same semaphore is shared among processes at different priority levels (use "HSemP" and "HSemV" in those cases). The **SemP** macro implements the "P" operation on a semaphore; it blocks the current process until the semaphore, **s**, is free. The **SemV** macro implements the complementary "V" operation, freeing the semaphore and starting the first process (if any), waiting in the semaphore queue. Note that the parameter to these functions is the semaphore proper, not a pointer to it. The **_SemP** and **_SemV** functions are primitives used by the macros (they should not be called directly).

The semaphore implementation is such that the routines may be used either as "binary" or "counting" semaphores.

Semaphores to be used by these routines must be initialized before use. This is automatically handled by the "SemAlloc" function, and should be performed manually if the semaphore is "statically" declared. See the "SemAlloc" function description for more information.

WARNING: THE SemP and SemV FUNCTIONS ARE IMPLEMENTED AS MACROS! THEY SHOULD NOT BE CALLED WITH A "s" PARAMETER WHOSE EVALUATION INVOLVES "SIDE EFFECTS", AS THE PARAMETER WILL BE EVALUATED MORE THAN ONCE!

For additional information, see the included paper by Jeff Mock: **Processes, Channels and Semaphores**.

RETURN VALUE

These functions return the "use" field from the **s** "Semaphore" structure.

RELATED FUNCTIONS

HSemP, HSemV, SemAlloc, SemFree

setjmp

initialize non-local jump

SYNOPSIS

```
#include <setjmp.h>

int setjmp(jmp_buf env)
```

DESCRIPTION

A call to the **setjmp** function saves a copy of the calling environment in the "jmp_buf" argument, **env**, for later use by a "longjmp" call. See the description of the "longjmp" function for more information.

RETURN VALUE

If the return is from the initial, direct, invocation of **setjmp**, zero is returned. If the return is a result of a subsequent call to the "longjmp" function, the return value is non-zero.

RELATED FUNCTIONS

longjmp

SetHiPriQ/SetLoPriQ

set queue pointers

SYNOPSIS

```
#include <conc.h>

void SetHiPriQ(void *front, void *back)

void SetLoPriQ(void *front, void *back)
```

DESCRIPTION

These functions provide "C" bindings for the Transputer "sthf/sthb" (**SetHiPriQ**), and "stlf/stlb" (**SetLoPriQ**), instructions. The **front** parameter is used to set the queue front pointer, the **back** parameter sets the queue back pointer. The values being set should be word aligned. See the appropriate INMOS documentation for more information about the "stXX" instructions.

These functions are implemented "inline". To get a functional version, precede your call with:

```
#undef SetHiPriQ
```

Or,

```
#undef SetLoPriQ
```

RETURN VALUE

None

RELATED FUNCTIONS

GetHiPriQ, GetLoPriQ

SetTime

set timer value for current priority

SYNOPSIS

#include <conc.h>

void SetTime(int time)

DESCRIPTION

The **SetTime** function sets the timer for the current priority level to **time**. Note that the timer runs at different rates for high (1 uS/tick), and low priority (64 uS/tick), processes.

This function is implemented "inline". To get a functional version, precede your call with:

#undef SetTime

For additional information see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

ProcAfter, ProcWait, Time

sin/sinf

compute sine

SYNOPSIS

#include <math.h>

double sin(double x)

float sinf(float x)

DESCRIPTION

The **sin** function returns the sine of **x** (measured in radians). The **sinf** function does the same thing for a single precision argument.

RETURN VALUE

A large magnitude argument may yield a result with little significance.

RELATED FUNCTIONS

acos, asin, atan, cos, tan

sinh/sinhf

compute hyperbolic sine

SYNOPSIS

```
#include <math.h>

double sinh(double x)

float sinhf(float x)
```

DESCRIPTION

The **sinh** function returns the hyperbolic sine of **x**. The **sinhf** function does the same thing for a single precision argument.

RETURN VALUE

A range error ("errno" is set to ERANGE), occurs if the magnitude of **x** is too large.

RELATED FUNCTIONS

cosh, tanh

sprintf

formatted write to memory

SYNOPSIS

```
#include <stdio.h>

int sprintf(char *ptr, const char *format, ...)
```

DESCRIPTION

The **sprintf** function writes characters into the array pointed to by **ptr**. The write is formatted under the control of the **format** argument, which specifies how subsequent arguments (if any), are to be converted. See the "printf" function description for more information about formatting options.

RETURN VALUE

The number of characters written into the array (not counting the terminating '\0' character).

RELATED FUNCTIONS

fprintf, printf, vfprintf, vprintf, vsprintf

sqrt/sqrtf

compute square root

SYNOPSIS

```
#include <math.h>

double sqrt(double x)

float sqrtf(float x)
```

DESCRIPTION

The **sqrt** function returns the floating point square root of a non-negative floating point argument **x**. The **sqrtf** function does the same thing for a single precision argument.

On floating point Transputers this function is implemented "inline". To get a functional version, precede your call with:

```
#undef sqrt
```

Or,

```
#undef sqrtf
```

RETURN VALUE

A domain error ("errno" is set to EDOM), occurs if **x** is negative.

RELATED FUNCTIONS

None

srand

set random number seed

SYNOPSIS

```
#include <stdlib.h>

void srand(unsigned int)
```

DESCRIPTION

The **srand** function sets the pseudo-random number "seed" value, for subsequent calls to "frand" and "rand". Explicitly calling **srand**, with the same value, allows restarting the same sequence. The initial setting of the "seed" is equivalent to calling **srand** with a seed value of one.

RETURN VALUE

None

RELATED FUNCTIONS

frand, rand

sscanf

formatted read from memory

SYNOPSIS

```
#include <stdio.h>

int sscanf(const char *ptr, const char *format, ...)
```

DESCRIPTION

The **sscanf** function reads characters from the array pointed to by **ptr**. The read is formatted under the control of the **format** argument, which specifies the legal input text sequences and conversion instructions. Subsequent arguments (if any), are used as pointers to the objects which receive converted data from the input. See the "scanf" function description for more information about formatting options.

RETURN VALUE

The **sscanf** function returns EOF if an input failure (running into the '\0' string terminator), occurs before any conversions. Otherwise, **sscanf** returns the number of input sequences which were matched, converted, and assigned. The number of sequences returned, may be less than the number provided for in the **format** string, if a match fails, or '\0' is encountered.

If running into '\0' occurs in the middle of matching an input sequence, such that the match may already be considered successful (not counting optionally matched whitespace), the '\0' is considered to merely terminate the current match. Any remaining conversions, however, will be immediately aborted if they require input.

RELATED FUNCTIONS

fscanf, scanf, strtod, strtol, strtoul

ssort

shell sort

SYNOPSIS

#include <stdlib.h>

void ssort(void *base, size_t nmemb, size_t size, int (*compare)(const void *, const void *))

DESCRIPTION

The **ssort** function is an implementation of the "shell sort" algorithm which has the same calling sequence as the standard "qsort" function. This function may be used to advantage for medium sized and large data sets which provoke worst case "qsort" performance (either almost already sorted or anti-sorted input data, depending on implementation). Note that for more arbitrary input data, "qsort" will almost always be faster.

See the "qsort" function description for detailed parameter explanations.

RETURN VALUE

None

RELATED FUNCTIONS

isort, qsort

strcat

string concatenation

SYNOPSIS

#include <string.h>

char *strcat(char *dst, const char *src)

DESCRIPTION

The **strcat** function appends the string pointed to by **src**, to the end of the string pointed to by **dst**. A copy of the first character of the **src** string overwrites the '\0' string terminator at the original end of the **dst** string; a copy of the second character in the **src** string is written in the next location beyond the end of **dst**, etc. The '\0' terminator in the **src** string is included in the copy to form a new **dst** string termination. If the resulting **dst** string overlaps the **src** string, the results are undefined.

RETURN VALUE

The **dst** address is returned.

RELATED FUNCTIONS

strncat

strchr

search string for character

SYNOPSIS

#include <string.h>

char *strchr(const char *ptr, int c)

DESCRIPTION

The **strchr** function searches a string, pointed to by **ptr**, for the first occurrence of the character **c**. A pointer to where the character was found is returned, or NULL, if a '\0' string terminator was found instead. The '\0' character is considered part of the string and may be searched for.

RETURN VALUE

See above

RELATED FUNCTIONS

strrchr

strcmp

string compare

SYNOPSIS

#include <string.h>

int strcmp(const char *s1, const char *s2)

DESCRIPTION

The **strcmp** function compares the string pointed to by **s1** and the string pointed to by **s2**.

RETURN VALUE

The **strcmp** function returns a value greater than zero if string **s1** is lexicographically greater than string **s2**. If string **s1** is equal to **s2**, a value of zero is returned. If string **s1** is lexicographically less than string **s2**, a negative value is returned.

RELATED FUNCTIONS

memcmp, strncmp

strcpy

string copy

SYNOPSIS

#include <string.h>

char *strcpy(char *dst, const char *src)

DESCRIPTION

The **strcpy** function copies the string pointed to by **src**, to the address pointed to by **dst**. The '\0' string termination is part of the string copied. If the resulting **dst** string overlaps the **src** string, the results are undefined.

This function is implemented as a macro and may be "inlined" depending on the value of **src**. To get a functional version, precede your call with:

#undef strcpy

RETURN VALUE

The **dst** address is returned.

RELATED FUNCTIONS

memcpy, memmove, strncpy

strcspn

character class complement substring search

SYNOPSIS

#include <string.h>

size_t strcspn(const char *s1, const char *s2)

DESCRIPTION

The **strcspn** function returns the length of the maximum initial segment of the string pointed to by **s1** which contains no characters from the string pointed to by **s2**.

RETURN VALUE

See above

RELATED FUNCTIONS

strpbrk, strspn

strerror

convert error into string

SYNOPSIS

```
#include <string.h>

char *strerror(int errnum)
```

DESCRIPTION

The **strerror** function takes an argument, **errnum**, which represents a value to which the "errno" variable could be set, and returns a string describing the error. The string buffer used is static, and may be overwritten by a subsequent call to **strerror**.

There is a basic problem when using this function (or "perror"), to translate the meaning of a value taken from "errno":

1. The value of "errno" may be the result of a host server I/O error; whereupon the host environment definitions for "errno" should be used.

2. The value of "errno" may be the result of an error detected by the runtime library on the Transputer; thus the **Transputer Toolset** "errno" definitions should be used.

The scheme actually adopted was to only report the "errno" values (and naming scheme), which are used by the library functions running on the Transputer. For errors which happen on the host system, the results of the **strerror** function aren't defined.

A further problem with "errno" arises because it is a shared global variable. In a program consisting of many processes, each of which uses the same "errno", how do you determine which process detected the error?

RETURN VALUE

A pointer to the error message.

RELATED FUNCTIONS

perror

strlen

string length

SYNOPSIS

#include <string.h>

size_t strlen(const char *ptr)

DESCRIPTION

The **strlen** function returns the length in bytes (not counting the '\0' terminator), of the string pointed to by **ptr**.

This function is implemented as a macro and may be "inlined" depending on the value of **ptr**. To get a functional version, precede your call with:

#undef strlen

RETURN VALUE

See above

RELATED FUNCTIONS

None

strncat

string concatenation with length

SYNOPSIS

```
#include <string.h>

char *strncat(char *dst, const char *src, size_t n)
```

DESCRIPTION

The **strncat** function appends up to **n** characters of the string pointed to by **src**, to the end of the string pointed to by **dst**. A copy of the first character of the **src** string overwrites the '\0' string terminator at the original end of the **dst** string; a copy of the second character in the **src** string is written in the next location beyond the end of **dst**, etc. A '\0' terminator is always written to the end of the new **dst** string. If the resulting **dst** string overlaps the **src** string, the results are undefined.

RETURN VALUE

The **dst** address is returned.

RELATED FUNCTIONS

strcat

strncmp

string compare with length

SYNOPSIS

#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n)

DESCRIPTION

The **strncmp** function compares up to **n** characters from the string pointed to by **s1** to the string pointed to by **s2**. The '\0' string terminator also stops the comparsion if the strings are shorter than **n** bytes in length.

RETURN VALUE

The return value of **strncmp** is based on at most the first **n** characters in the strings being compared. The **strncmp** function returns a value greater than zero if string **s1** is lexicographically greater than string **s2**. If string **s1** is equal to **s2**, a value of zero is returned. If string **s1** is lexicographically less than string **s2**, a negative value is returned.

RELATED FUNCTIONS

memcmp, strcmp

strncpy

string copy with length

SYNOPSIS

```
#include <string.h>

char *strncpy(char *dst, const char *src, size_t n)
```

DESCRIPTION

The **strncpy** function copies up to **n** characters from the string pointed to by **src**, to the address pointed to by **dst**. The '\0' string termination is part of the string copied and is counted against the maximum length **n**. If the resulting **dst** string overlaps the **src** string, the results are undefined. If the **src** string was shorter than **n** bytes in length (counting terminator), '\0' padding bytes are appended to the **dst** string until **n** bytes total have been written.

RETURN VALUE

The **dst** address is returned.

RELATED FUNCTIONS

memcpy, memmove, strcpy

strpbrk

character class string search

SYNOPSIS

#include <string.h>

char *strpbrk(const char *s1, const char *s2)

DESCRIPTION

The **strpbrk** function finds the first occurrence, in the string pointed to by **s1**, of any character in the string pointed to by **s2**.

RETURN VALUE

A pointer to the character found, or NULL if no character from **s2** is found in **s1**.

RELATED FUNCTIONS

memchr, strchr

strrchr

reverse search string for character

SYNOPSIS

#include <string.h>

char *strrchr(const char *ptr, int c)

DESCRIPTION

The **strrchr** function searches a string, pointed to by **ptr**, for the last occurrence of the character **c**. A pointer to where the character was found is returned, or NULL, if a '\0' string terminator was found instead. The '\0' character is considered part of the string and may be searched for.

RETURN VALUE

See above

RELATED FUNCTIONS

strchr

strspn

character class substring search

SYNOPSIS

#include <string.h>

size_t strspn(const char *s1, const char *s2)

DESCRIPTION

The **strspn** function returns the length of the maximum initial segment of the string pointed to by **s1** which contains only characters from the string pointed to by **s2**.

RETURN VALUE

See above

RELATED FUNCTIONS

strcspn, strpbrk

strstr

substring search

SYNOPSIS

```
#include <string.h>

char *strstr(const char *s1, const char *s2)
```

DESCRIPTION

The **strstr** function returns a pointer to the first occurrence of the string pointed to by **s2**, in the string pointed to by **s1**. The '\0' string terminator is not counted as part of the search.

RETURN VALUE

On success, a pointer to the matching string found in the string pointed to by **s1**. On failure, a NULL pointer is returned. If **s2** is of zero length, s1 is returned.

RELATED FUNCTIONS

strcspn, strpbrk, strspn

strtod/strtol/strtoul

convert strings to numbers

SYNOPSIS

```
#include <stdlib.h>

double strtod(const char *nptr, char **endptr)

long strtol(const char *nptr, char **endptr, int base)

unsigned long strtoul(const char *nptr, char **endptr, int base)
```

DESCRIPTION

The **strtod** function converts an ASCII numeric string pointed to by **nptr** into the equivalent floating point number. The **strtol** and **strtoul** functions do the same thing for long integers, and unsigned long integers, respectively. In all cases, whitespace is allowed before the numeric string and the first unrecognized character (in the selected **base**), stops the conversion. Also, if **endptr** is non-NULL, a pointer to the unrecognized character is stored into the object pointed to by **endptr**.

strtod recognizes a numeric string consisting of an optional sign, a digit sequence with optional decimal point, and an optional exponent ('e' or 'E', optional sign and exponent digit sequence).

strtol and **strtoul** recognize integral numbers consisting of an optional sign and digit sequence in a specified **base**. If the value of **base** is zero, the expected digit sequence is that of a "C" integer constant (ie. decimal, hexadecimal, or octal notation, without any integer suffix). If the value of **base** is between 2 and 36, the expected digit sequence is a sequence of numbers and letters representing an integer with radix specified by **base**. The letters from 'a' ('A'), through 'z' ('Z'), are used for the values 10 to 35 when **base** is greater that 10; only digits and letters whose value is less than the **base** are allowed. If the value of **base** is 16, the "0x" and 0X" hexadecimal prefixes are permitted.

RETURN VALUE

The converted value. If no conversion could be performed (or the correct value would cause underflow), zero is returned. If the correct value would cause overflow **strtod** returns +/- HUGE_VAL, **strtol** returns LONG_MAX/LONG_MIN, and **strtoul** returns ULONG_MAX. If underflow or overflow is detected, "errno" is set to ERANGE.

RELATED FUNCTIONS

atof, atoi, atol

strtok

break string into tokens

SYNOPSIS

```
#include <string.h>

char *strtok(char *s1, const char *s2)
```

DESCRIPTION

The **strtok** function returns pointers into the string pointed to by **s1**, which are delimited by characters in the string pointed to by **s2**. The **strtok** function is designed to be used repetitively to decompose a string into individual '\0' terminated substrings. The **strtok** function remembers, between calls, where it was in processing the string pointed to by **s1**. This, the **s1** argument should be replaced with a NULL pointer for the second and subsequent calls to **strtok** for the same (**s1**), string. The delimiter string, **s2**, may vary from one call to the next.

The first call to **strtok** returns a NULL pointer if the first character in the string pointed to by **s1**, is also in the delimiter string, **s2**. If not, **strtok** searches for the first character in **s1** which is in **s2**, replaces it with a '\0' terminator, and returns the pointer **s1**.

Each subsequent call behaves in a similar fashion, except, the search begins at the first character after the '\0' terminator written by the previous call to **strtok**.

RETURN VALUE

A pointer to the start of the next token, or NULL if no token matching the delimiting characters could be found.

RELATED FUNCTIONS

None

system

call host command processor

SYNOPSIS

```
#include <stdlib.h>

int system(const char *ptr)
```

DESCRIPTION

The **system** function passes the string pointed to by **ptr**, to the host environment command processor for execution. If **ptr** is NULL, the **system** function may be used to determine if a command processor exists.

RETURN VALUE

If **ptr** was NULL, a non-zero return value indicates a command processor exists. If **ptr** was non-NULL (a string), the return value is assumed to be a value returned from the command processor after execution of the supplied string.

RELATED FUNCTIONS

exit

tan/tanf

compute tangent

SYNOPSIS

#include <math.h>

double tan(double x)

float tanf(float x)

DESCRIPTION

The **tan** function returns the tangent of **x** (measured in radians). The **tanf** function does the same thing for a single precision argument.

RETURN VALUE

A large magnitude argument may yield a result with little significance.

RELATED FUNCTIONS

acos, asin, atan, cos, sin

tanh/tanhf

compute hyperbolic tangent

SYNOPSIS

```
#include <math.h>

double tanh(double x)

float tanhf(float x)
```

DESCRIPTION

The **tanh** function returns the hyperbolic tangent of **x**. The **tanhf** function does the same thing for a single precision argument.

RETURN VALUE

See above

RELATED FUNCTIONS

cosh, sinh

time

determine calendar time

SYNOPSIS

#include <time.h>

time_t time(time_t *timer)

DESCRIPTION

The **time** function determines the current calendar time.

RETURN VALUE

The **time** function returns the best approximation to the current time the host system can compute. If **timer** is not a NULL pointer, the return value is also stored in the object it points to.

RELATED FUNCTIONS

asctime, ctime, gmtime, localtime

Time

get timer value for current priority

SYNOPSIS

```
#include <conc.h>

int Time(void)
```

DESCRIPTION

The **Time** function atomically returns the value of the timer for the current priority level. Note that the timer runs at different rates for high (1 uS/tick), and low priority (64 uS/tick), processes.

This function is implemented "inline". To get a functional version, precede your call with:

```
#undef Time
```

For additional information see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

See above

RELATED FUNCTIONS

ProcAfter, ProcWait, SetTime

tmpfile

create temporary file

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile(void)
```

DESCRIPTION

The **tmpfile** function creates a temporary binary file which is automatically deleted when it is closed (or the program terminates). The file is opened in the "wb+" mode (see the "fopen" function description for more information).

RETURN VALUE

A pointer to the opened stream, or NULL if the file couldn't be created.

RELATED FUNCTIONS

fopen, tmpnam

tmpnam

create unique filename

SYNOPSIS

```
#include <stdio.h>

char *tmpnam(char *ptr)
```

DESCRIPTION

The **tmpnam** function creates a filename which is designed to not conflict with existing files. The **tmpnam** function generates a different name each time it is called, up to at least TMP_MAX times.

If the argument, **ptr**, is non-NULL, the generated name is stored starting at the location it points to. The return value will be set to **ptr**.

If **ptr** is NULL, the return value will point to an internal buffer area, which may be overwritten on subsequent **tmpnam** calls.

There is an inherent race condition in using this function involving the time between the **tmpnam** call and the subsequent file creation call. We suggest you use the "tmpfile" function for all new work.

RETURN VALUE

See above

RELATED FUNCTIONS

tmpfile

toascii/tolower/toupper

map characters

SYNOPSIS

#include <ctype.h>

int toascii(int c)

int tolower(int c)

int toupper(int c)

DESCRIPTION

These macros/functions map various classes of characters into other classes. The argument, **c**, is the character to be mapped. These macros/functions are designed to evaluate the argument only once, and are safe for use with arguments with side-effects. The individual descriptions:

toascii - Map **c** into the ASCII character set (**c** & 0x7F).

tolower - If **c** is a upper case character, convert it to the lower case equivalent. Leaves other characters unchanged.

toupper - If **c** is a lower case character, convert to to the upper case equivalent. Leaves other characters unchanged.

RETURN VALUE

The (possibly), converted character.

RELATED FUNCTIONS

isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

ungetc

push back a character

SYNOPSIS

#include <stdio.h>

int ungetc(int c, FILE *stream)

DESCRIPTION

The **ungetc** function pushes the character specified by c, back onto the input stream. Only one ungetc operation is guaranteed to work without an intervening read operation (although some host systems will support more). Characters pushed back onto a stream are read in the reverse order they were pushed.

Use of one of the file positioning functions ("fseek", "rewind", ...), will cause any pushed back characters to be discarded. A successful call to "ungetc" will clear the EOF flag for the stream. EOF may not be pushed back.

RETURN VALUE

The **ungetc** function returns the character pushed back (or EOF on error).

RELATED FUNCTIONS

fgetc, fputc, getc, getchar, putc, putchar

unlink

delete file

SYNOPSIS

```
#include <stdio.h>

int unlink(const char *pathname)
```

DESCRIPTION

The **unlink** function deletes the file specified by **pathname**. If the file is currently open the results are undefined.

The **unlink** function is not part of the ANSI standard library. We recommend use of the equivalent "remove" function for new work.

RETURN VALUE

If the deletion is successful, zero is returned; otherwise non-zero.

RELATED FUNCTIONS

remove, rename

va_arg

variable argument access macro

SYNOPSIS

#include <stdarg.h>

type va_arg(va_list ap, type_expression)

DESCRIPTION

The **va_arg** macro expands to an expression which has the **type** and value of the next argument in the argument list managed by the "va_list" structure, **ap**. Each invocation of **va_arg** modifies **ap** so that successive arguments are returned. The "va_list" structure, **ap**, must have been initialized by a call to "va_start" prior to any use of **va_arg**. The parameter, **type_expression**, is a type name specified such that appending a '*' to the parameter will yield a legal "C" type expression which is a pointer to the specified **type**.

See the description of the "va_end" macro for further information (and an example of use).

RETURN VALUE

The value of the next argument in the variable argument list (assumed to be of **type** type).

RELATED FUNCTIONS

va_end, va_start

va_end

variable argument termination macro

SYNOPSIS

#include <stdarg.h>

void va_end(va_list ap)

DESCRIPTION

The **va_end** macro terminates processing of the arguments in a variable argument list, **ap**. This macro should be called after the argument list is initialized by a call to "va_start", and has been accessed through a series of "va_arg" calls. The example shows the usage of the three macros.

RETURN VALUE

None

RELATED FUNCTIONS

va_arg, va_start

EXAMPLE

The function, "argtbl", builds an array of pointers to strings which represents the parameters passed to "argtbl". The argument array is then passed to function "argtbl2" for further processing:

```
     #include <stdarg.h>

     #define MAX_ARGS 30 /* Whatever the application
requires */

     void argtbl(int num_ptrs, ...)
          {
          va_list ap;
          char *arg_ptrs[MAX_ARGS];
          int ptr_num = 0;

          if(num_ptrs > MAX_ARGS)
               num_ptrs = MAX_ARGS;
          va_start(ap, num_ptrs);
          while(ptr_num < num_ptrs)
               arg_ptrs[ptr_num++] = va_arg(ap, char *);
          va_end(ap);
          argtbl2(num_ptrs, arg_ptrs);
          }
```

Note that in this example, each call to "argtbl" must have a first parameter which defines how many following parameters are in the argument list.

va_start

variable argument initialization macro

SYNOPSIS

```
#include <stdarg.h>

void va_start(va_list ap, right_param)
```

DESCRIPTION

The **va_start** macro initializes the "va_list" structure, **ap**, such that subsequent calls to the "va_arg" macro will evaluate to the arguments in the enclosing function parameter list. These parameters will be located immediately after the last listed formal parameter, **right_param** (right before the ", ...").

See the description of the "va_end" macro for further information (and an example of use).

RETURN VALUE

None

RELATED FUNCTIONS

va_arg, va_end

VChan

get virtual channel pointer

SYNOPSIS

```
#include <conc.h>

Channel *VChan(int lchan)
```

DESCRIPTION

The **VChan** function translates a logical channel number (**lchan**), into the corresponding virtual channel pointer. The virtual channel pointer which is returned may then be used with the various virtual channel I/O routines.

The user selected **lchan** value can range from 6 to 32767. It must be the same value used to refer to the same channel (on a given node), in the "Network Information File" to be used with LD-NET. LD-NET uses the information during loading to configure and route virtual channels. **VChan** is basically a mechanism to allow the application program to get a pointer to a virtual channel that LD-NET sets up during the loading process.

Whenever possible, smaller logical channel numbers should be used since it minimizes the memory overhead of the virtual channel control information. See the LD-NET documentation for additional information about logical channels and associated memory usage.

Channel pointers returned by **VChan** may only be used with the various virtual channel I/O routines. Using a virtual channel pointer with a non-virtual channel I/O routine will not work, and may "crash" the Transputer node.

RETURN VALUE

A pointer to the corresponding virtual channel. If **lnum** is outside the range of 6 to 32767 (or wasn't mentioned in the "Network Information File" used by LD-NET to load this node), NULL will be returned.

RELATED FUNCTIONS

VChanIn, VChanInChar, VChanInInt, VChanOut, VChanOutChar, VChanOutInt, VChanVin, VChanVOut, VProc*

VChanIn/VChanInChar/VChanInInt
reading from virtual channels

SYNOPSIS

```
#include <conc.h>

void VChanIn(Channel *c, void *ptr, int n)

char VChanInChar(Channel *c)

int VChanInInt(Channel *c)
```

DESCRIPTION

The **VChanIn** function reads **n** bytes of data, from the virtual channel pointed to by **c**, to the buffer pointed to by **ptr**. The **VChanInChar** and **VChanInInt** functions may be used to read, and return, the value of a byte or word, respectively, read from the virtual channel pointed to by **c**.

These functions are analogous to the "ChanIn", "ChanInChar" and "ChanInInt" functions (see descriptions elsewhere in this manual), but these functions support both virtual and regular ("soft"), channels. When used with regular channels the two sets of functions are equivalent.

When used with a virtual channel, the **VChanIn** function allows the receipt of a zero length message. This "synchronization-only" behavior is not allowed with regular channels on T2/T4/T8 processors. When receiving zero length messages, the specific destination buffer pointer value (**ptr**), is unimportant. A value of NULL is a good choice since it helps document what is going on.

All messages read using one of these functions must be matched by a corresponding virtual channel output function call whose message length EXACTLY matches. If variable length data transfer is needed, you should first send a message header specifying the size of the variable length data to come, followed by sending the actual data. This allows the input and output operations to always have matching message lengths. See the "VChanVIn" and "VChanVOut" functions for another approach to variable length virtual channel communication.

Since virtual channels are bi-directional, it is perfectly permissable to have processes concurrently performing input and output on the same "end" of a virtual channel. Since each "end" of a virtual channel is independent, you can potentially have four processes working on the same virtual channel (although perhaps not all on the same node since the "ends" may be on different nodes). As regular T2/T4/T8 channels are uni-directional you may only have one input and one output process working on the same channel, even if you use them with these virtual channel input functions.

For additional information about "regular channel" primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

See above

RELATED FUNCTIONS

VChan, VChanOut, VChanOutChar, VChanOutInt, VChanVIn, VChanVOut, VProc*

VChanOut/VChanOutChar/VChanOutInt

writing to virtual channels

SYNOPSIS

```
#include <conc.h>

void VChanOut(Channel *c, void *ptr, int n)

void VChanOutChar(Channel *c, int byte)

void VChanOutInt(Channel *c, int word)
```

DESCRIPTION

The **VChanOut** function writes **n** bytes of data, to the virtual channel pointed to by **c**, from the buffer pointed to by **ptr**. The **VChanOutChar** and **VChanOutInt** functions may be used to write the value of a **byte** or **word**, respectively, to the virtual channel pointed to by **c**.

When used with a virtual channel, the **VChanOut** function allows the transmission of a zero length message. This "synchronization-only" behavior is not allowed with regular channels on T2/T4/T8 processors. When sending zero length messages, the specific source buffer pointer value (**ptr**), is unimportant. A value of NULL is a good choice since it helps document what is going on.

See the discussion for the corresponding input operations ("VChanIn" ...), for a more lengthy introduction.

For additional information about "regular channel" primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

VChan, VChanIn, VChanInChar, VChanInInt, VChanVIn, VChanVOut, VProc*

VChanVIn/VChanVOut

variable length virtual channel I/O

SYNOPSIS

```
#include <conc.h>

int VChanVIn(Channel *c, void *ptr, int n)

void VChanVOut(Channel *c, void *ptr, int n)
```

DESCRIPTION

The **VChanVIn** function reads at most **n** bytes of data, from the virtual channel pointed to by **c**, to the buffer pointed to by **ptr**. The return value is the number of bytes actually read, or -1 if the message is longer than **n** bytes. The **VChanVOut** function writes **n** bytes of data, to the virtual channel pointed to by **c**, from the buffer pointed to by **ptr**.

These functions are similar to "VChanIn"/"VChanOut", except for the variable length capability. These functions should only be used with virtual channels since regular T2/T4/T8 "soft" channels do not support this capability. For proper operation, messages to be received by the **VChanVin** function should always be sent by the **VChanVOut** function (and vice-versa).

Note that it is perfectly permissable for the output message length specified to **VChanVOut**, or the maximum input message length specified to **VChanVIn**, to be zero. This provides the capability of a "synchronization-only" message. When receiving or sending zero length messages, the specific buffer pointer value (**ptr**), is unimportant. A value of NULL is a good choice since it helps document what is going on.

See the "VChanIn" function description for more information about virtual channel usage.

RETURN VALUE

See above

RELATED FUNCTIONS

VChan, VChanIn, VChanInChar, VChanInInt, VChanOut, VChanOutChar, VChanOutInt, VProc*

vfprintf

vararg formatted write to stream

SYNOPSIS

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

int vfprintf(FILE *stream, const char *format, va_list arg)
```

DESCRIPTION

The **vfprintf** function is identical to the "fprintf" function, except it uses a "packaged" argument structure, instead of a variable length argument list. The "packaged" argument structure is initialized by a "va_start" macro call, followed by an optional "va_arg" macro call for each argument to be added. The **vfprintf** function does not invoke the "va_end" macro. See the descriptions of the "va_start", "va_arg", and "va_end" macros, for more information.

RETURN VALUE

The number of characters written (negative values indicate an error).

RELATED FUNCTIONS

fprintf, printf, sprintf, va_arg, va_end, va_start, vprintf, vsprintf

vprintf

vararg formatted write

SYNOPSIS

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

int vprintf(const char *format, va_list arg)
```

DESCRIPTION

The **vprintf** function is identical to the "printf" function, except it uses a "packaged" argument structure, instead of a variable length argument list. The "packaged" argument structure is initialized by a "va_start" macro call, followed by an optional "va_arg" macro call for each argument to be added. The **vprintf** function does not invoke the "va_end" macro. See the descriptions of the "va_start", "va_arg", and "va_end" macros, for more information.

RETURN VALUE

The number of characters written (negative values indicate an error).

RELATED FUNCTIONS

fprintf, printf, sprintf, va_arg, va_end, va_start, vfprintf, vsprintf

VProcAlt*

determine the status of virtual channels

VProcAlt
VProcAltList
VProcSkipAlt
VProcSkipAltList
VProcTimerAlt
VProcTimerAltList

SYNOPSIS

```
#include <conc.h>

int VProcAlt(Channel *c1, ...)

int VProcAltList(Channel **clist)

int VProcSkipAlt(Channel *c1, ...)

int VProcSkipAltList(Channel **clist)

int VProcTimerAlt(int time, Channel *c1, ...)

int VProcTimerAltList(int time, Channel **clist)
```

DESCRIPTION

These functions allow a process to determine the status of one or more input virtual channels. On T2/T4/T8 processors these functions may also be used with regular "soft" channels with the restriction that the associated input must be of non-zero length. These functions operate correctly with either zero or non-zero length input messages when used with virtual channels.

VProcAlt, **VProcSkipAlt**, and **VProcTimerAlt** take an explicit NULL terminated list of pointers to channels as parmeters. **VProcAltList**, **VProcSkipAltList**, and **VProcTimerAltList** take a NULL terminated array of pointers to channels as a parameter. Note that the channel pointers are prioritized based on the order they are listed in.

VProcAlt and **VProcAltList** cause the current process to block until one of the channels in its argument list is ready for input. On completion, the routine returns a zero based index into the parameter list for the ready channel.

VProcSkipAlt and **VProcSkipAltList** check specified channels. If one of the channels is ready for input, a zero based index into the parameter list is returned, otherwise (-1) is returned. These routines do not block waiting for one of the channels, they return immediately.

VProcTimerAlt and **VProcTimerAltList** block the current process until one of the channels is ready for input or the value of the clock (as read by the "Time" function), is after the **time** parameter. If the routine times out, a -1 is returned, otherwise a zero based index into the parameter list is returned.

These functions are equivalent to the "ProcAlt*" functions described elsewhere in this manual, except they can handle both regular and virtual channels. For additional information on the non-virtual channel versions (which is also applicable to these functions), see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

See above

RELATED FUNCTIONS

ProcAfter, ProcWait, Time, VChan, VChanIn, VChanInChar, VChanInInt, VChanOut, VChanOutChar, VChanOutInt, VChanVin, VChanVOut

vsprintf

vararg formatted write to memory

SYNOPSIS

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

int vsprintf(char *ptr, const char *format, va_list arg)
```

DESCRIPTION

The **vsprintf** is identical to the "sprintf" function, except it uses a "packaged" argument structure, instead of a variable length argument list. The "packaged" argument structure is initialized by a "va_start" macro call, followed by an optional "va_arg" macro call for each argument to be added. The **vsprintf** function does not invoke the "va_end" macro. See the descriptions of the "va_start", "va_arg", and "va_end" macros, for more information.

RETURN VALUE

The number of characters written (not counting the '\0' terminating character).

RELATED FUNCTIONS

fprintf, printf, sprintf, va_arg, va_end, va_start, vfprintf, vprintf

write

low level file write

SYNOPSIS

#include <stdio.h>

int write(int handle, char *buf, size_t n)

DESCRIPTION

The **write** function writes **n** bytes of data, to the file opened with **handle**, from the buffer pointed to by **buf**.

The file must have been previously opened by a call to "open" or "creat", and be writable.

As this function is not supported in the ANSI standard library, its use in new work is discouraged (see the "fopen", "fclose", "fread", and "fwrite" functions for the recommended replacements).

RETURN VALUE

The **write** function returns the number of bytes actually written. A return value of (-1) indicates an error and "errno" will be set appropriately.

Note the difficulty that writing a 65535 byte file on a computer with a 16 bit "int" size causes (the non-error and error return values are the same).

RELATED FUNCTIONS

close, creat, dup, dup2, open, read