malloc

allocate heap memory

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size)
```

DESCRIPTION

The **malloc** function allocates a region of heap memory large enough to hold an object of **size** bytes.

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 allocated region.

RELATED FUNCTIONS

addfree, calloc, cfree, free, realloc

memccpy

string copy with length and terminator

SYNOPSIS

#include <string.h>

void *memccpy(const void *dst, const void *src, int c, size_t n)

DESCRIPTION

The **memccpy** function copies bytes from **src** to **dst**, until either **n** bytes have been copied, or a character equal to **c** is copied, whichever happens first.

RETURN VALUE

If the character **c** is copied, the return value will be the next address after the copied character in the **dst** string. If **c** is not found, the copy is terminated because of the maximum length **n**, and a NULL pointer is returned.

RELATED FUNCTIONS

memccpy, memcpy, strcpy

memchr

string search with length

SYNOPSIS

#include <string.h>

void *memchr(const void *ptr, int c, size_t n)

DESCRIPTION

The **memchr** function searches at most **n** bytes, starting at **ptr**, for the byte value **c**.

RETURN VALUE

If the character **c** is found, the return value will be a pointer to it. If **c** is not found, NULL is returned.

RELATED FUNCTIONS

memccpy

memcmp

compare bytes

SYNOPSIS

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n)
```

DESCRIPTION

The **memcmp** function compares byte string **s1** against byte string **s2** for **n** bytes, returning zero if they are identical, a negative integer if string **s1** is lexicographically less than string **s2**, and a positive integer otherwise.

RETURN VALUE

See above

RELATED FUNCTIONS

bcmp

memcpy

string copy with length

SYNOPSIS

#include <string.h>

void *memcpy(void *dst, const void *src, size_t n)

DESCRIPTION

The **memcpy** function copies **n** bytes from the location pointed to by **src** to the location pointed to by **dst**.

The **memcpy** function is equivalent to "bcopy", but with a reversed **src** and **dst** argument order.

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

#undef memcpy

RETURN VALUE

Returns the value of **dst**.

RELATED FUNCTIONS

bcopy, memmove, strcpy

memmove

overlapping string copy with length

SYNOPSIS

#include <string.h>

void *memmove(void *dst, const void *src, size_t n)

DESCRIPTION

The **memmove** function copies **n** bytes from the location pointed to by **src** to the location pointed to by **dst**. The **memmove** function is like "memcpy", except it correctly handles copying where the **src** and **dst** strings overlap.

RETURN VALUE

Returns the value of **dst**.

RELATED FUNCTIONS

bcopy, memcpy, strcpy

memset

string initialize

SYNOPSIS

#include <string.h>

void *memset(void *ptr, int c, size_t n)

DESCRIPTION

The **memset** function writes **n** copies of the character **c** into memory starting at **ptr**.

RETURN VALUE

Returns the value of **ptr**.

RELATED FUNCTIONS

bcopy, memcpy, memmove, strcpy

modf/modff

truncate floating point number

SYNOPSIS

```
#include <math.h>

double modf(double x, double *y)

float modff(float x, float *y)
```

DESCRIPTION

The **modf** function breaks a floating point number, **x**, into integral and fractional parts. Both components retain the same sign as **x**. The integral part is stored as a floating point number at the address pointed to by **y**; the fractional part is returned as the value of the function. The **modff** function does the same thing for single precision arguments.

RETURN VALUE

See above.

RELATED FUNCTIONS

None

Move2D/Move2DNonZero/Move2DZero
2D block move

SYNOPSIS

```
#include <conc.h>

void Move2D(void *src, void *dst, int w, int n, int sw, int dw)

void Move2DNonZero(void *src, void *dst, int w, int n, int sw, int dw)

void Move2DZero(void *src, void *dst, int w, int n, int sw, int dw)
```

DESCRIPTION

These functions provide "C" bindings for the two dimensional block move instructions supported by some Transputers. The functions are only available on Transputers which support the instructions.

The arguments to all three functions are identical:

src - The source base address for the move.

dst - The destination base address for the move.

w - The width in bytes of the rows to be copied.

n - The number of rows to be copied.

sw - The source array "stride" width in bytes.

dw - The destination array "stride" width in bytes.

The **Move2D** function uses the "move2dinit" and "move2dall" instructions to perform a two dimensional block move, without worrying about the values of the bytes moved.

The **Move2DNonZero** function is the same as **Move2D**, except only bytes which are non-zero are copied from the source to the destination (uses the "move2dnonzero" instruction instead of "move2dall").

The **Move2DZero** function is the same as **Move2D**, except only bytes which are zero are copied from the source to the destination (uses the "move2dzero" instruction instead of "move2dall").

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

#undef Move2D

Or,

#undef Move2DNonZero

Or,

#undef Move2DZero

See the applicable INMOS documentation for more information about the underlying instructions used by these functions.

RETURN VALUE

None

RELATED FUNCTIONS

None

_ns_exit

non-server exit

SYNOPSIS

```
#include <stdlib.h>

void _ns_exit(void)
```

DESCRIPTION

The **_ns_exit** function is an analog to "exit" which is used when a program running on the Transputer isn't connected to a host server. Under these circumstances, (there is no environment to return a status value to), this function simply stops the current process.

This function is one of three functions which are provided for use when no host server is desired. The others:

"_ns_main" - A replacement for "_main" (specified to TLNK as the desired entry point). Does everything the normal "_main" does except request command line arguments from the host.

"_ns_printf" - Equivalent in capabilities to "_printf", except no I/O server is assumed. All output is written to the "_host_chan_out" "hard" channel as plain ASCII text (no protocol).

There are a number of reasons to use these functions in lieu of the full fledged equivalents:

1. They are very simple, and are useful during the early stages of a port of the **Transputer Toolset** (when not everything might be working correctly).

2. They use a very small amount of ram. This is useful when the program being run doesn't require anything beyond "_printf" for I/O, and you wish to get the last ounce of performance out of it by loading everything possible in on-chip ram. A related case is when hardware development is underway, and only on-chip ram is currently functioning (or that is all the design allows the Transputer).

To use these three functions, they must replace the normal protocol-driven host I/O services. This is done by specifying "_ns_main" as the entry point to TLNK, and ensuring that no routine which requires the host server is called during program execution.

RETURN VALUE

None

RELATED FUNCTIONS

_ns_printf

_ns_printf

non-server _printf

SYNOPSIS

#include <stdio.h>

void _ns_printf(const char *format, ...)

DESCRIPTION

The **_ns_printf** function is an analog to "_printf" which is used when a program running on the Transputer isn't connected to a host server. All output is written to the "_host_chan_out" "hard" channel as plain ASCII text (no protocol).

This function is one of three functions which are provided for use when no host server is desired. The others:

"_ns_main" - A replacement for "_main" (specified to TLNK as the desired entry point). Does everything the normal "_main" does except request command line arguments from the host.

"_ns_exit" - Like, "exit", but doesn't require host server.

There are a number of reasons to use these functions in lieu of the full fledged equivalents:

1. They are very simple, and are useful during the early stages of a port of the **Transputer Toolset** (when not everything might be working correctly).

2. They use a very small amount of ram. This is useful when the program being run doesn't require anything beyond "_printf" for I/O, and you wish to get the last ounce of performance out of it by loading everything possible in on-chip ram. A related case is when hardware development is underway, and only on-chip ram is currently functioning (or that is all the design allows the Transputer).

To use these three functions, they must replace the normal protocol-driven host I/O services. This is done by specifying "_ns_main" as the entry point to TLNK, and ensuring that no routine which requires the host server is called during program execution.

WARNING: UNLIKE THE REGULAR "_printf", THIS FUNCTION IS NOT SAFE TO USE CONCURRENTLY. RESTRICT YOUR USE OF "_ns_printf" TO A SINGLE PROCESS.

RETURN VALUE

None

RELATED FUNCTIONS

_ns_exit

open

low level file open

SYNOPSIS

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

int open(char *path, int flags, int mode)
```

DESCRIPTION

The **open** function opens the file specified by **path**. The allowed operations on the file are controlled by the setting of **flags**, and the **mode** parameter is used to specify the desired access permissions when **open** is told to create the file.

The following symbolic bit mask definitions are used with the **flags** parameter:

O_APPEND - Position the file pointer to the end of the file prior to performing a write.
O_BINARY - Open the file in untranslated mode (no CR/LF mapping nonsense).
O_CREAT - If the file doesn't exist create it and open it for writing. See the following description for the mode **flag**.
O_EXCL - Return an error if the file exists (used with O_CREAT).
O_NONBLOCK - Return if the I/O would block.
O_RDONLY - Open file only for reading.
O_RDWR - Open the file for reading and writing.
O_TEXT - Open the file in translated mode (perform CR/LF mapping if host OS requires it).
O_TRUNC - Open the file and toss contents.
O_WRONLY - Open file only for writing.

If the values of the above macros are bitwise OR'ed together, and passed as the **flags** parameter, the actions are presumed to be combined (not all combinations are legal however).

Note that not all of the macro definitions are usable with all the host operating systems which exist. In particular, the O_BINARY and O_TEXT flags are commonly restricted to PC's and VMS, while the O_NONBLOCK flag is used mostly with UNIX and derivatives. If a flag has no meaning in a particular environment it may still be used in programs without any adverse effects.

The **mode** parameter applies only when the O_CREAT bit is part of the **flags** argument, and determines what permission setting the file should be given when closed (see the "creat" function description).

The following symbolic bit mask macro definitions are allowed with **mode**:

S_IREAD - Allow file reading
S_IWRITE - Allow file writing

If the values of the above two macros are bitwise OR'ed together, and passed as the **mode** parameter, the file is assumed to allow both reading and writing.

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 **open** function returns a positive value (a "handle"), for the file if the open/create is successful, otherwise (-1) is returned and "errno" is set appropriately.

RELATED FUNCTIONS

close, creat, dup, dup2, read, write

perror

write "errno" message

SYNOPSIS

```
#include <stdio.h>

void perror(const char *ptr)
```

DESCRIPTION

The **perror** function writes the error message corresponding to the current value of "errno" to the "stderr" output stream. The error message printed is composed of the string pointed to by **ptr** (if non-NULL), followed by a colon and space, and the error message which corresponds to the value of "errno".

There is a basic problem when using this function (or "strerror"), to print the meaning of a value in "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 a result of a error detected with the runtime library on the Transputer; thus the **Transputer Toolset** "errno" definitions should be used.

The solution 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 **perror** 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

None

RELATED FUNCTIONS

strerror

PFork/PForkHigh/PForkLow

process forking

SYNOPSIS

void PFork(Forkblk f, PDes p)

void PForkHigh(Forkblk f, PDes p)

void PForkLow(Forkblk f, PDes p)

DESCRIPTION

These three routines start a new process at either the current priority (**PFork**), high priority (**PForkHigh**), or low priority (**PForkLow**). These routines are part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**. The routines take two parameters; a "Forkblk" structure **f**, and a process descriptor structure **p**.

The "Forkblk" structure, **f**, must have been previously initialized with a call to "PForkInit". The "PDes" structure, **p**, must have been previously initialized with a call to "PSetup". The forked process terminates by returning from the initial entry function specified to "PSetup" when **p** was initialized.

WARNING: THESE FUNCTIONS ARE IMPLEMENTED AS MACROS! THEY SHOULD NOT BE CALLED WITH A "p" PARAMETER WHOSE EVALUATION INVOLVES "SIDE EFFECTS", AS THE PARAMETER WILL BE EVALUATED MORE THAN ONCE! DOING SOMETHING LIKE:

```
PFork(f,PSetup(...stuff...));
```

WILL NOT WORK! DO THE FOLLOWING INSTEAD:

```
PDes p;
...
p = PSetup(...stuff...);
PFork(f,p);
```

See the **Transputer Concurrency (Fork/Join Model)** section at the beginning of this manual for more information.

RETURN VALUE

These routines have no return value and are implemented as macros.

RELATED FUNCTIONS

PForkInit, PHalt, PJoin, PRun, PSetup, PStop

PForkInit
initialize process fork structure

SYNOPSIS

void PForkInit(Forkblk f, int n)

DESCRIPTION

The **PForkInit** macro initializes a "Forkblk" structure. This function is part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**. The function takes two parameters; a "Forkblk" structure to initialize, **f**, and the number of processes which will be forked, **n**.

The fork/join operation is similar to using one of the "ProcPar*" functions, except the originating parent process continues execution in parallel with the forked child. This avoids the allocation of an extra stack (workspace), and also the need for an initial entry function for the extra process. The parameter, **n**, thus specifies the total number of outstanding processes which will be synchronized with the later "PJoin" call (including the original parent process). After the "PJoin" call (which will block the parent until all child processes terminate), the parent process will awake and continue execution. Note that all the child processes need not be forked at the same time. If the number of children to be forked is not known in advance, the number can be adjusted at run time with the proviso that it always remain positive. For example, the following program skeleton shows one approach to adjusting the number dynamically:

```
PForkInit(f,1);
for( ... )
     {
     ...
     p = PSetup(...)
     pri = ProcToHigh();
     f.count++;
     if(pri)
          (void) ProcToLow();
     PFork(f,p);
     ...
     }
PJoin(f);
```

WARNING: THIS FUNCTION IS IMPLEMENTED AS A MACRO! IT SHOULD NOT BE CALLED WITH A "f" PARAMETER WHOSE EVALUATION INVOLVES "SIDE EFFECTS", AS THE PARAMETER WILL BE EVALUATED MORE THAN ONCE!

See the **Transputer Concurrency (Fork/Join Model)** section at the start of this manual for more information.

RETURN VALUE

This routine has no return value and is implemented as a macro.

RELATED FUNCTIONS

PFork, PForkHigh, PForkLow, PHalt, PJoin, PRun, PSetup, PStop

PHalt

kill current process and save descriptor

SYNOPSIS

```
#include <conc.h>

void PHalt(PDes *pid)
```

DESCRIPTION

The **PHalt** function kills the current process and stores the terminated process descriptor in **pid**. Since the process descriptor is saved, the process may be later restarted using "PRun". This function is part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**.

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

```
#undef PHalt
```

RETURN VALUE

None

RELATED FUNCTIONS

PFork, PForkHigh, PForkInit, PForkLow, PJoin, PRun, PSetup, PStop

PJoin

merge previously forked processes

SYNOPSIS

```
#include <conc.h>

void PJoin(Forkblk *f)
```

DESCRIPTION

The **PJoin** function blocks until all processes which are associated with the "Forkblk" structure (**f**), terminate. The "Forkblk" structure is assumed to have been initialized by a call to "PForkInit", and subsequent process fork actions (calls to "PFork", "PForkHigh" or "PForkLow"). This function is part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**.

See the **Transputer Concurrency (Fork/Join Model)** section at the start of this manual for more information.

RETURN VALUE

None

RELATED FUNCTIONS

PFork, PForkHigh, PForkInit, PForkLow, PHalt, PRun, PSetup, PStop

pow/powf

compute x to the y power

SYNOPSIS

```
#include <math.h>

double pow(double x, double y)

float powf(float x, float y)
```

DESCRIPTION

The **pow** function returns the the value of **x** raised to the power **y**. The **powf** function does the same thing for single precision arguments.

RETURN VALUE

See above. A domain error occurs if **x** is negative and **y** is non-integral ("errno" is set to EDOM). A range error ("errno" is set to ERANGE), may also occur.

RELATED FUNCTIONS

exp, log, log10

printf

formatted write

SYNOPSIS

```
#include <stdio.h>

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

DESCRIPTION

The **printf** function writes to the "stdout" stream. The write is formatted under the control of the **format** argument, which specifies how subsequent arguments (if any), are to be converted for output. 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. The **printf** function returns when it finishes evaluating the **format** string.

The **format** string consists of a '\0' terminated character sequence composed of ordinary characters (other than '%'), which are simply copied to "stdout". The '%' character is used to mark the beginning of a format specification.

Format Specification

A format specification is generally used to describe how an argument to "printf" should be converted, formatted, and written to "stdout". After the '%' character, the following fields make up a format specification:

1. Zero or more "flags". These are general qualifiers to the format specification.

2. An optional, non-zero, decimal integer specifying a minimum field width for the result. If the converted and formatted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the "left adjustment" flag, described later, has been given), to the minimum field width.

3. An optional "precision" qualifier which gives the minimum number of digits to appear for the 'd', 'i', 'o', 'u', 'x', and 'X' conversions, the number of digits to appear after the decimal point for 'e', 'E', and 'f' conversions, the maximum number of significant digits for the 'g' and 'G' conversions, or the maximum number of characters to be written from a string with the 's' conversion. The "precision" qualifier takes the form of a period, followed by an optional decimal integer (zero is assumed if the integer is omitted).

4. An optional 'h' qualifier indicating that the argument to the following 'd', 'i', 'o', 'u', 'x', or 'X' conversion will be a "short int" or "unsigned short int" (or that the argument to a 'n' conversion will be a "short int" pointer). In place of the 'h', a 'l' may appear, which has the same general meaning as 'h', except all arguments are assumed to be "long", rather than "short". The final optional alternative to the 'h' and 'l' qualifiers is 'L', which indicates the argument to the following 'e', 'E', 'f', 'g', or 'G' conversion will be of type "long double", rather than "double". If the 'h', 'l', or 'L' qualifiers is used with a conversion other than one of the ones listed, the results are undefined.

5. Finally, the character which specifies which conversion is to be performed.

Note that either the minimum field width, or the "precision" (or both), may use '*' in place of the decimal integer, to allow the values to be dynamically configured. The '*' is replaced with the value of an additional argument to **printf** (at run time). If either '*' option is used, corresponding integer argument(s) must be supplied to **printf** (in order), before the argument which is the subject of the conversion. A negative field width argument is taken as a '-' flag, followed by a positive field width. A negative precision is ignored.

Flags

The legal "flags" are:

'-' - Left justify the conversion within the field.

'+' - Force a sign to be written for signed conversions.

' ' - Prepend a space before the result for signed conversions if the result was non-negative (and a '+' sign wouldn't normally be printed). The '+' flag takes precedence over this flag if both are used.

'#' - Use an "alternate" form for the conversion. For 'o', this increases the precision to force the first digit of the result to be zero. For 'x' or 'X', a non-zero result will have '0x' or '0X' (respectively), prepended. For the 'e', 'E', 'f', 'g', and 'G' conversions, the result will always have a decimal point (the decimal point is not normally written unless at least one digit follows it). For the 'g' and 'G' conversions, trailing zeros (otherwise pruned), will be written. The '#' flag produces undefined results with conversions other than these.

'0' - For the 'd', 'i', 'o', 'u', 'x', 'X', 'e', 'E', 'f', 'g', and 'G' conversions, leading zeros (following any base or sign indication), are used to pad the result to the field width. This is in lieu of the padding with spaces which would normally occur. The '-' flag takes precedence over the '0' flag. If an explicit precision is specified for the 'd, 'i', 'o', 'u', 'x', and 'X' conversions, the '0' flag is also ignored. The '0' flag produces undefined results with conversions other than these.

Conversions

The following are the legal conversion characters:

'c' - A "int" argument is converted to "unsigned char", and the resulting character written out.

'd'
'i'
'o'
'u'
'x'
'X' - A "int" argument is converted to signed decimal ('d' or 'i'), unsigned octal ('o'), unsigned decimal ('u'), or unsigned hexadecimal ('x' or 'X'). The letters "abcdef" are used for the six additional digits needed with 'x' ("ABCDEF" with 'X'). The "precision" qualifier specifies the minimum number of digits to appear (expanded, if necessary, with leading zeros). If no "precision" is specified, the default is one digit. If zero is converted with a precision of zero, nothing will be printed.

'e'
'E' - A "double" argument is converted in the style [-]d.dddeSdd ('S' represents a '+' or '-' sign). The 'E' form will substitute 'E' for the 'e' exponent field character. The number of digits after the decimal point is equal to the "precision". If no "precision" is specified, the default is six digits. If the "precision" is zero, no decimal point is written. The exponent always contains at elast two digits. The result is rounded to the number of digits written.

'f' - A "double" argument is converted to decimal notation in the style [-]ddd.ddd, where the number of digits after the decimal point is equal to the "precision". If no "precision" is specified, the default is six digits. If the "precision" is zero, no decimal point is written. If a decimal point appears, it will be followed by at least one digit. The result is rounded to the number of digits written.

'g'
'G' - A "double" argument, to be converted with 'g', is equivalent to using either 'f' or 'e', depending on the characteristics of the value being converted. If the 'G' conversion is used, the conversion is equivalent to using 'E'. The "precision" specifies the number of significant digits (minimum one). The 'e' conversion will only be used if the exponent is less than (-4), or greater than (or equal to), the "precision". Trailing zeros are removed from the fractional part of the result. A decimal point only appears if followed by a digit.

'n' - A "integer" pointer argument is assumed. The current number of characters written (so far), by this **printf** call, is stored at the address specified in the argument.

'p' - A "void" pointer is displayed in an implementation dependent manner (hexadecimal notation for the **Transputer Toolset**).

's' - The argument is assumed to be a pointer to an array of characters (a '\0' terminated "string"). No more characters are written than the "precision" (if specified).

'%' - A '%' is written to "stdout" (ie. a "%%" format specification writes a single '%').

Conversion specifications other than these produce undefined results.

Don't Worry

In no case does a small field width cause truncation of a result! If the result of a conversion is wider than the field, the field width is expanded. There is, however, a maximum field width (per-conversion), of 509 characters in this implementation.

RETURN VALUE

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

RELATED FUNCTIONS

fprintf, sprintf, vfprintf, vprintf, vsprintf

EXAMPLES

To print a date and time in the form "Saturday, June 2, 10:01", where "weekday" and "month" are pointers to '\0' terminated strings:

```
#include <stdio.h>

print_date(char *weekday, char *month, int day, int hour, int min)
        {
        printf("%s, %s %d, %.2d:%.2d\n", weekday, month, day, hour,
min);
        }
```

To print PI to 4 decimal places:

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

main()
        {
        printf("pi=%.4f", 4.0 * atan(1.0));
        }
```

_printf

simple formatted write

SYNOPSIS

#include <stdio.h>

int _printf(const char *format, ...)

DESCRIPTION

The **_printf** function is a stripped down version of the normal "printf" function. It is mainly used where the full facilities of "printf" are not required, and the vastly smaller amount of memory consumed is helpful. It doesn't support floating point, option "flags", field widths or precision values. It also doesn't support any return value (despite the prototype). See the "printf" function description for more information.

RETURN VALUE

None

RELATED FUNCTIONS

printf

ProcAfter

suspend process until specified time

SYNOPSIS

```
#include <conc.h>

void ProcAfter(int time)
```

DESCRIPTION

The **ProcAfter** function blocks the current process until the value of the timer for the current priority level is after **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 ProcAfter
```

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

RETURN VALUE

None

RELATED FUNCTIONS

ProcWait, SetTime, Time

ProcAlloc

allocate asynchronous process

SYNOPSIS

```
#include <conc.h>

Process *ProcAlloc(void (*f)(), int s, int np, int p1, int p2, ...)
```

DESCRIPTION

The **ProcAlloc** function builds a stack frame for a new process using memory obtained from the heap. It also allocates memory for a "Process" structure which will define the stack frame characteristics. The stack frame is set to a total size of **s** bytes, and is initialized with the values of **np** words worth of integral parameters (**p1**, **p2**, ...). The initial function to execute in the new process is set using function pointer **f**; it will be passed the parameters (**process_ptr**, **p1**, **p2**, ...), upon process creation. The **process_ptr** parameter will be of type "Process *" and will be the address of the "Process" structure returned by the **ProcAlloc** call.

The **s** (stack size), parameter may be set to zero, in which case 64K bytes will be allocated on 32 bit Transputers (2K bytes on 16 bit Transputers). Remember that the default heap size is only 128K bytes (32K bytes on the 16 bit Transputers), and counting overhead, only one 64K region may be successfully allocated on a 32 bit Transputer. Modification of the default heap size will allow allocation up to the limits of physical memory availability (see the **Heap Management** section at the beginning of this manual for more information). The actual amount of stack space required will vary based on the demands the process places on "auto" variables, and the nesting depth of function calls, etc. Keep in mind, for example, that the "printf" family of library functions requires 600 or 700 bytes of stack space! There is also a "minimum" stack size specification which is allowed based on having space to set up the function linkage, and store the parameters, and the requested allocation will be increased to this "minimum" if necessary.

On successful allocation, **ProcAlloc** returns a pointer to an initialized "Process" structure (memory for which it has also allocated from the heap). This structure may be later used with one of the process creation functions such as "ProcRun", "ProcRunHigh", "ProcRunLow", "ProcPar", "ProcParList", or "ProcPriPar".

When (or if), a process allocated in this fashion is no longer in use, the memory used may be returned to the heap using the "ProcFree" function. If it is just desired to change the parameters in the process, a call to "ProcParam" may be used (without requiring returning the old stack frame and getting a new one).

Note that if a more "static" approach to stack frame creation is desired, the "ProcInit" function should be considered in lieu of **ProcAlloc**. If "ProcInit" is used, the "Process" structure may also be statically allocated and the use of the heap entirely avoided. The "Fork/Join" functions ("PFork", "PJoin", PSetup", "PRun", ...), could also be used to completely eliminate the need for a "Process" structure.

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

RETURN VALUE

The **ProcAlloc** function returns NULL if it was unable to allocate the requested stack frame. Otherwise, a pointer to the initialized "Process" structure is returned.

RELATED FUNCTIONS

PFork, PForkHigh, PForkInit, PForkLow, PJoin, ProcFree, ProcInit, ProcPar, ProcParam, ProcParList, ProcPriPar, ProcRun, ProcRunHigh, ProcRunLow, PRun, PSetup

ProcAlt*

determine the status of channels

ProcAlt
ProcAltList
ProcSkipAlt
ProcSkipAltList
ProcTimerAlt
ProcTimerAltList

SYNOPSIS

```
#include <conc.h>

int ProcAlt(Channel *c1, ...)

int ProcAltList(Channel **clist)

int ProcSkipAlt(Channel *c1, ...)

int ProcSkipAltList(Channel **clist)

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

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

DESCRIPTION

These functions allow a process to determine the status of one or more Transputer input channels.

ProcAlt, **ProcSkipAlt**, and **ProcTimerAlt** take an explicit NULL terminated list of pointers to channels as parmeters. **ProcAltList**, **ProcSkipAltList**, and **ProcTimerAltList** 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.

ProcAlt and **ProcAltList** 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.

ProcSkipAlt and **ProcSkipAltList** 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.

ProcTimerAlt and **ProcTimerAltList** 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.

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

RETURN VALUE

See above

RELATED FUNCTIONS

ChanIn, ChanInChar, ChanInInt, ChanOut, ChanOutChar, ChanOutInt, ProcAfter, ProcWait, Time

ProcCall

call a function with a new workspace

SYNOPSIS

```
#include <conc.h>

int ProcCall(int (*func), void *ws, int wssize, int p1, ...)
```

DESCRIPTION

The **ProcCall** function allows you to call a function and specify the region of memory it should use for its workspace (stack). A pointer to the function to call is passed as **func**, the start of the new workspace to use as **ws**, and the workspace size as **wssize** (in bytes). Up to five integer parameters may be passed to the function (**p1**, ...).

Because the Transputer has internal ram which is MUCH faster to access than normal ram, this function has been implemented to allow the programmer to dynamically utilize this scarce resource. Moving the workspace of computationally intensive functions into the fast ram is usually a good first cut at improving execution speed.

It is not uncommon for "C" programs to execute 40% faster when the workspace is moved into internal memory. The overhead of **ProcCall** is about 5 times the overhead of a normal function call.

The fast ram may be used in a static fashion for the stack (as well as program and data storage), by directives to the TLNK linker. See the **TLNK Transputer Linker User Guide** for more information about these possibilities.

For additional information see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**. Note that the "internal" keyword used in the example in his paper is not needed with the **Transputer Toolset**, and is replaced by directives to TLNK.

RETURN VALUE

The value returned is that produced by the call to **func**.

RELATED FUNCTIONS

None

EXAMPLE

Call a function named "compute" with one parameter (the integer value 13):

```
#include <conc.h>

#define SIZE 512
char stack[SIZE];
extern int compute();

main()
      {
      int result;

      result = ProcCall(compute, stack, SIZE, 13);
      }
```

In this case, "compute" is executed with its stack stored in 512 bytes of memory (the "stack" array). Often, the array reference is replaced by the physical address of the desired region in internal ram. You must ensure that the region you specify is both, unused during the execution time of this function, and large enough for the stack space the function (and any nested calls), require.

ProcGetPriority

get current process priority level

SYNOPSIS

```
#include <conc.h>

int ProcGetPriority(void)
```

DESCRIPTION

The **ProcGetPriority** function atomically returns the current priority level.

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

```
#undef ProcGetPriority
```

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

RETURN VALUE

0 if the current process is high priority, 1 if low.

RELATED FUNCTIONS

ProcToLow, ProcToHigh

ProcFree

deallocate asynchronous process

SYNOPSIS

```
#include <conc.h>

void ProcFree(Process *p)
```

DESCRIPTION

The **ProcFree** function returns a "Process" structure pointed to by **p** (and the associated stack frame it defines), to the heap free memory pool. The structure pointed to by **p** MUST have been originally obtained by a call to "ProcAlloc", or the results are undefined. You should also ensure that no currently running (or ever executable), process is using the stack frame being deallocated.

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

RETURN VALUE

None

RELATED FUNCTIONS

ProcAlloc, ProcInit, ProcPar, ProcParam, ProcParList, ProcPriPar, ProcRun, ProcRunHigh, ProcRunLow

ProcInit

initialize asynchronous process

SYNOPSIS

```
#include <conc.h>

void ProcInit(Process *p, void (*f)(), void *sp, int s, int np, int p1, int p2, ...)
```

DESCRIPTION

The **ProcInit** function takes a pointer **p** to a "Process" structure, and related information about the process, and initializes the "Process" structure accordingly. The required process information is:

1. A pointer **f** to the function to call upon process creation.

2. A stack frame pointed to by **sp**, which is **s** bytes long.

3. A list of **np** words worth of integral parameters to be passed to the initial function (**p1**, **p2**, ...).

The **ProcInit** function is called by "ProcAlloc" to initialize the stack frames it allocates from the heap. **ProcInit** may be called directly when you wish to use memory obtained elsewhere for the stack frame and "Process" structure.

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

RETURN VALUE

None

RELATED FUNCTIONS

ProcAlloc, ProcFree, ProcPar, ProcParam, ProcParList, ProcPriPar, ProcRun, ProcRunHigh, ProcRunLow

ProcPar/ProcParList

run asynchronous processes

SYNOPSIS

```
#include <conc.h>

void ProcPar(Process *p1, Process *p2, ...)

void ProcParList(Process **plist)
```

DESCRIPTION

The **ProcPar** function takes a NULL terminated parameter list of "Process" pointers, and runs all the associated processes (**p1**, **p2**, ...), in parallel. Control is returned to the calling function when all the initiated processes terminate. The **ProcParList** function does the same thing but uses a pointer to a NULL terminated vector of "Process" pointers (**plist**), as a parameter instead.

In either case, the processes are run at the same priority as the calling function.

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

RETURN VALUE

None

RELATED FUNCTIONS

ProcAlloc, ProcFree, ProcInit, ProcParam, ProcPriPar, ProcRun, ProcRunHigh, ProcRunLow

ProcParam

modify asynchronous process parameters

SYNOPSIS

```
#include <conc.h>

void ProcParam(Process *p, int p1, int p2, ...)
```

DESCRIPTION

The **ProcParam** function takes a pointer **p** to a "Process" structure, and a set of integral parameters (**p1**, **p2**, ...), for the function to be called by that process, and initializes the process stack frame so that the function will get the parameters when the process is created. The "Process" structure must have been previously initialized by either "ProcAlloc" or "ProcInit" for correct operation. The **ProcParam** function is used when it is desired to re-use an existing "Process" structure and stack frame and just change the parameters to the function call. The stack frame must not be in use by a running process or the results are undefined.

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

RETURN VALUE

None

RELATED FUNCTIONS

ProcAlloc, ProcFree, ProcInit, ProcPar, ProcParList, ProcPriPar, ProcRun, ProcRunHigh, ProcRunLow

ProcPriPar

run mixed priority processes

SYNOPSIS

```
#include <conc.h>

void ProcPriPar(Process *phigh, Process *plow)
```

DESCRIPTION

The **ProcPriPar** function takes two "Process" pointer arguments, and runs the associated processes in parallel. The first process argument (**phigh**), is run as a high priority process, the second (**plow**), is run at low priority. Control is returned to the calling function when both processes terminate.

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

RETURN VALUE

None

RELATED FUNCTIONS

ProcAlloc, ProcFree, ProcInit, ProcPar, ProcParam, ProcParList, ProcRun, ProcRunHigh, ProcRunLow

ProcReschedule

suspend and delay current process

SYNOPSIS

```
#include <conc.h>

void ProcReschedule(void)
```

DESCRIPTION

The **ProcReschedule** function blocks the current process and places it at the end of the process queue at the current priority. This function may be used to implement a "busy wait" mechanism while giving other processes a chance to execute. Where possible, you should avoid this function in favor of one of the more efficient, and elegant, approaches supported by the Transputer hardware. For example, if you need to wait on input data on a channel, use one of the "ProcAlt" family of functions instead.

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

```
#undef ProcReschedule
```

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

RETURN VALUE

None

RELATED FUNCTIONS

None

ProcRun/ProcRunHigh/ProcRunLow

run asynchronous process

SYNOPSIS

```
#include <conc.h>

void ProcRun(Process *p)

void ProcRunHigh(Process *p)

void ProcRunLow(Process *p)
```

DESCRIPTION

These functions create a new process using the information in the "Process" structure pointed to by **p**.

The **ProcRun** function causes the new process to have the same priority as the process which calls **ProcRun**. The **ProcRunHigh** function forces the process to execute at high priority, while **ProcRunLow** forces low priority execution.

The "Process" structure must have been previously initialized by a call to "ProcAlloc" or "ProcInit" (possibly modified by a call to "ProcParam").

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

RETURN VALUE

None

RELATED FUNCTIONS

ProcAlloc, ProcInit, ProcPar, ProcParam, ProcParList, ProcPriPar

ProcStop

kill current process

SYNOPSIS

#include <conc.h>

void ProcStop(void)

DESCRIPTION

The **ProcStop** function kills the current process. As the process descriptor information is not saved (see the "Phalt" function), there is no way to later restart a process killed in this fashion.

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

#undef ProcStop

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

This function is equivalent to the "PStop" function.

RETURN VALUE

None

RELATED FUNCTIONS

PHalt, PStop

ProcToHigh

change to high priority

SYNOPSIS

```
#include <conc.h>

int ProcToHigh(void)
```

DESCRIPTION

The **ProcToHigh** function forces the current process to execute at high priority. It is designed to be used with the "ProcToLow" function to protect critical code sections.

RETURN VALUE

The previous priority level is returned: zero if it was already at high priority, one if it was at low priority.

RELATED FUNCTIONS

ProcToLow

ProcToLow

change to low priority

SYNOPSIS

```
#include <conc.h>

int ProcToLow(void)
```

DESCRIPTION

The **ProcToLow** function forces the current process to execute at low priority. It is designed to be used with the "ProcToHigh" function to protect critical code sections.

RETURN VALUE

The previous priority level is returned: one if it was already at low priority, zero if it was at high priority.

RELATED FUNCTIONS

ProcToHigh

ProcWait

suspend process for specified time

SYNOPSIS

```
#include <conc.h>

void ProcWait(int time)
```

DESCRIPTION

The **ProcWait** function blocks the current process for at least **time** timer ticks. 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 ProcWait
```

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

RETURN VALUE

None

RELATED FUNCTIONS

ProcAfter, SetTime, Time

PRun

process fork primitive

SYNOPSIS

#include <conc.h>

void PRun(PDes p)

DESCRIPTION

The **PRun** function takes an initialized "PDes" structure, **p**, and starts the associated process. This function is part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**, and is a primitive used by the "PFork", "PForkHigh", and "PForkLow" macros. Boiled down, this function is a "C" binding for the Transputer "runp" instruction.

The "PDes" structure, **p**, must have been initialized with a previous call to the "PSetup" function. Note that "PSetup" doesn't provide the low order priority bit (if needed). To run a process at low priority you execute something like:

```
PRun(PSetup(...) | 1);
```

Other process descriptors which may be used with **PRun** come from functions such as "PHalt", "ChanReset", etc. See the appropriate INMOS documentation concerning the "runp" instruction for more information.

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

#undef PRun

See the **Transputer Concurrency (Fork/Join Model)** section at the start of this manual for more information.

RETURN VALUE

None

RELATED FUNCTIONS

PFork, PForkHigh, PForkInit, PForkLow, PHalt, PJoin, PSetup, PStop

PSetup

initialize process fork process

SYNOPSIS

#include <conc.h>

PDes PSetup(void *ws, void (*func)(), int wsize, int psize, ...)

DESCRIPTION

The **PSetup** function initializes a workspace and process descriptor for used with a later "process fork" operation ("PFork", "PForkHigh" or "PForkLow"). The "PDes" structure returned is an initialized process descriptor which points to the new "forkable" process created by **PSetup**. The **PSetup** function is analogous to "ProcInit", except the "Process" structure is not required. The parameters include a pointer to the memory area to use for the new process workspace (stack), **ws**, the size of the workspace in bytes, **wsize**, a pointer to the function to serve as entry point for the new process, **func**, and a specification of how many words worth of memory the parameters to be passed to the function will occupy, **psize**. If **psize** is non-zero, the desired parameters should follow **psize** in the function argument list.

The **PSetup** function is part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**. See the **Transputer Concurrency (Fork/Join Model)** section at the start of this manual for more information.

RETURN VALUE

See above

RELATED FUNCTIONS

PFork, PForkHigh, PForkInit, PForkLow, PHalt, PJoin, PRun, PStop

PStop

kill current process

SYNOPSIS

```
#include <conc.h>

void PStop(void)
```

DESCRIPTION

The **PStop** function kills the current process. As the process descriptor information is not saved (see the "PHalt" function), there is no way to later restart a process killed in this fashion. This function is part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**.

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

```
#undef PStop
```

This function is equivalent to the "ProcStop" function.

RETURN VALUE

None

RELATED FUNCTIONS

PFork, PForkHigh, PForkInit, PForkLow, PHalt, PJoin, PRun, PSetup

putc/putchar

write a character

SYNOPSIS

```
#include <stdio.h>

int putc(int c, FILE *stream)

int putchar(int c)
```

DESCRIPTION

The **putc** macro writes the character specified by c to the output **stream**. Since **putc** is defined to be a macro, the arguments may be evaluated more than once, and must not have side-effects. To get a functional version, use either the "fputc" equivalent, or precede the function call with:

```
#undef putc
```

The putchar macro is identical to putc, but without the stream argument ("stdout" is assumed). The same precautions relative to argument side effects (and solutions), apply.

RETURN VALUE

These functions return the character written (or EOF on error).

RELATED FUNCTIONS

fgetc, fputc, getc, getchar, ungetc

puts

write a line

SYNOPSIS

#include <stdio.h>

int puts(const char *ptr)

DESCRIPTION

The **puts** function writes the string pointed to by **ptr** to the "stdout" stream. A newline is written to replace the terminating '\0' character.

RETURN VALUE

The **puts** function returns EOF if an error occurs; otherwise a non-negative value.

RELATED FUNCTIONS

fgets, fputs, gets