IV. C Machine Interface Library

NAME

Conventions - of the C machine interface library

FUNCTION

The functions and variables documented in this section are usable just like any of those in Section II or Section III, but need not be known to the typical C programmer. Rather, they are called upon by higher level functions to perform machine dependent operations, to provide machine dependent information, or merely to provide an important service with efficiency and/or extra precision.

They are isolated in a separate section a) to avoid cluttering an already extensive collection of useful functions with arcana, and b) to show prospective implementors what is required in the way of low level support for a new machine. Note that Section III serves much the same purpose for implementors of new operating system interfaces.

NAME

_addexp - scale double exponent

SYNOPSIS

```
DOUBLE _addexp(d, n, msg)
    DOUBLE d;
    COUNT n;
    TEXT *msg;
```

FUNCTION

_addexp effectively multiplies the double d by two raised to the power n, although it endeavors to do so by some speedy ruse. If the double result is too large in magnitude to be represented by the machine, _range is called with msg.

RETURNS

_addexp returns the double result d * (1 << n), or any value returned by _range.

EXAMPLE

```
DOUBLE sqrt(x)
    DOUBLE x;
    {
    COUNT n;

    n = _unpack(&x);
    x = newton(x);
    if (n & 1)
        x =* SQRT2;
    return (_addexp(x, n >> 1, "can't happen"));
    }
```

SEE ALSO

_frac, _range, _unpack

NAME

_domain - report domain error

SYNOPSIS

```
VOID _domain(msg)
    TEXT *msg;
```

FUNCTION

_domain is called by math functions to report a domain error, i.e., the fact that an input value lies outside the set of values over which the function is defined. It copies msg to _domerr, then calls _raise for the condition _domerr. This exception, if not caught, results in an error exit that prints the NUL terminated string at msg to STDERR, followed by a newline.

There is no way of inhibiting domain errors, though any code using _when to handle them may choose to ignore their occurrence.

RETURNS

_domain never returns to its caller. It may return from an instance of _when that is willing to handle a domain error; otherwise the program exits, reporting failure.

EXAMPLE

```
DOUBLE sqrt(x)
    DOUBLE x;
    {
    if (x < 0)
        _domain("negative argument to sqrt");
    ...
```

SEE ALSO

_domerr, _raise, _range, _when

NAME

_domerr - domain error condition

SYNOPSIS

TEXT *_domerr

FUNCTION

_domerr is the condition raised when a domain error occurs, i.e., when a math function discovers that an input value lies outside the set of values over which the function is defined.

SEE ALSO

_domain, _raise, _ranerr

NAME

_dtens - powers of ten

SYNOPSIS

```
DOUBLE _dtens[];
```

FUNCTION

_dtens is an array of doubles with values 1, 10, 100, 10**4, 10**8, etc. up to the largest such number the machine can represent. The number of entries in _dtens is recorded in the variable _ntens.

SEE ALSO

_ntens

NAME

_dzero - double zero

SYNOPSIS

```
DOUBLE _dzero;
```

FUNCTION

_dzero is a double zero, provided for convenience more than necessity.

SEE ALSO

_huge, _tiny

NAME

_fcan - canonicalize floating point datum

SYNOPSIS

```
COUNT _fcan(pd)
    TEXT *pd;
```

FUNCTION

_fcan is a machine dependent routine required by the C code generators to translate native double floating data to a canonical format. Each code generator can then translate from canonical to target machine format, irrespective of the host environment.

The canonical form is an array of eight characters stored in place of the double number at pd. pd[0] is zero if the number is positive, else 0200; pd[1] is the most significant byte of the fraction, with an assumed binary point to the left of its most significant bit; the remaining fraction bytes are stored in descending order of significance at pd[2] through pd[7]. If the number is nonzero, the most significant (0200) bit of pd[1] is set, so that the fraction is in the half-open interval [1/2, 1).

It is assumed that the number at pd is normalized on entry to _fcan.

RETURNS

_fcan returns the power of two by which the fraction must be multiplied to give the proper value. The sign and fraction bytes are written in place of the double number.

NAME

_frac - extract integer from fraction part

SYNOPSIS

```
COUNT _frac(pd, mul)
    DOUBLE *pd, mul;
```

FUNCTION

_frac forms the double product of *pd and mul, then partitions it into an integer plus a double fraction in the interval [-1/2, 1/2], delivers the fractional part to *pd and the low bits of the integer part as the value of the function. If the integer part cannot be properly represented as a COUNT, it is truncated on the left without remark.

RETURNS

_frac returns the low bits of the integer part of the product (*pd * mul) as the value of the function and writes the fractional part of the product at *pd.

EXAMPLE

```
DOUBLE sind(x)
    DOUBLE x;
    {
    COUNT n;

    n = _frac(&x, 1.0/90.0);
    ...
```

SEE ALSO

_addexp, _unpack

NAME

_huge - largest double number

SYNOPSIS

DOUBLE _huge

FUNCTION

_huge is the largest representable double number.

SEE ALSO

_dzero, _tiny

NAME

_norm - convert double to normalized text string

SYNOPSIS

```
COUNT _norm(s, d, prec)
    TEXT *s;
    DOUBLE d;
    BYTES prec;
```

FUNCTION

_norm factors the double d into a) a double in the interval [0.1, 1) or zero, and b) an integral power of ten. The first prec digits of the fraction are written as text characters in the buffer starting at s. If the number is negative on entry, it is forced positive.

RETURNS

The value of the function on return is the power of ten to which the fraction string in s must be raised to give the value of d. If d is zero, all characters in s are '0's and the value returned is zero.

SEE ALSO

_round

NAME

_ntens - number of powers of ten

SYNOPSIS

```
COUNT _ntens;
```

FUNCTION

_ntens is the number of elements in the array _dtens, which holds various powers of ten as double numbers.

SEE ALSO

_dtens

NAME

_poly - compute polynomial

SYNOPSIS

```
DOUBLE _poly(d, tab, n)
    DOUBLE d, *tab;
    COUNT n;
```

FUNCTION

_poly computes the polynomial of order n in the independent variable d, using the coefficients in the table pointed to by tab. Horner's method is used, taking tab[0] as the coefficient of the highest power of d, so the value computed is:

```
tab[n] + d * (tab[n-1] + d * (... + d * tab[0]))
```

No precautions are taken against overflow or underflow.

RETURNS

_poly returns the double value of the polynomial of order n in d.

EXAMPLE

```
return (x * _poly(x * x, coeffs, 6));
```

NAME

_raise - raise an exception

SYNOPSIS

```
VOID _raise(ptr, cx)
    TEXT **ptr, **cx;
```

FUNCTION

_raise signals the presence of a condition that must be handled by an earlier call to _when. The _when/_raise mechanism is used to perform a broad spectrum of stack manipulations normally beyond the scope of the C language, including: Ada exception handling, Pascal nonlocal goto's, Idris process switching, editor interrupt fielding, and math error reporting.

The handler to be first considered is specified by ptr. If ptr is -1 or NULL, the latest _when call is used as the start of a search for a willing handler; otherwise ptr must have been set by an earlier _when call to specify that call as the starting point of the search.

If cx is NULL or -1, then the first handler encountered returns to its caller with the value zero; otherwise cx must match a condition argument of one of the registered handlers to be considered, or at some level it must be handled by a NULL terminating a list of condition arguments.

The return from _when caused by a _raise call cleans up the stack if either ptr or cx is NULL. Otherwise, the handler for that _when call remains on the stack and is made the latest of the chain of handlers.

RETURNS

_raise never returns to its caller. It returns from the latest willing _when call with registers, stack, and handler chain restored to that level; the value returned by _when is nonnegative. The handler chain is initialized to a single catchall handler which calls error to print an error message, and takes an error exit. If the condition can be interpreted as the address of a pointer to a NUL terminated string, then that string, followed by a newline, is used as the error message; otherwise the message is "unchecked condition".

EXAMPLE

To exit on end of file:

```
TEXT *endfile {"unchecked end of file"};

VOID readrec(buf)
    TEXT *buf;
    {
    if (fread(STDIN, buf, 80) != 80)
        _raise(NULL, &endfile);
    }
 ...
    switch(_when(NULL, &endfile, NULL))
        {
    case 1:
```

```
        oneof();
        }
```

SEE ALSO

_when, error(II), enter(II), leave(II)

BUGS

You are not expected to understand this.

NAME

_ranerr - range error condition

SYNOPSIS

```
TEXT *_ranerr
```

FUNCTION

_ranerr is the condition raised when a range error occurs, i.e., when a math routine discovers that a return value is too large to represent. Unlike most conditions, the range condition may be inhibited from time to time by writing a nonzero value in _ranerr.

SEE ALSO

_domerr, _range

NAME

_range - report range error

SYNOPSIS

```
DOUBLE _range(msg)
    TEXT *msg;
```

FUNCTION

_range is called by math functions to report a range error, i.e., the production of an output value that cannot be represented properly by the machine. If _ranerr is NULL, _range copies msg to _ranerr, then calls _raise for the condition _ranerr. This exception, if not caught, results in an error exit that prints the NUL terminated string at msg to STDERR, followed by a newline.

If _ranerr is not NULL, the condition is not raised, and _range returns to its caller.

RETURNS

If _range returns to its caller, the value returned is the largest double that can be represented by the machine; otherwise the _ranerr condition is raised and _range does not return to its caller. It may return from an instance of _when that is willing to handle a range error; otherwise the program exits, reporting failure.

EXAMPLE

```
if (_lnhuge < x)
    _range("exp overflow");
```

SEE ALSO

_domain, _ranerr, _raise, _when

NAME

_round - round off a fraction string

SYNOPSIS

```
COUNT _round(s, n, prec)
    TEXT *s;
    BYTES n, prec;
```

FUNCTION

_round rewrites the n character buffer starting at s as a properly rounded string of prec digits. If prec is outside the buffer, or if (s[prec] < '5'), no action is taken. Otherwise, the next character to the left is incremented and carries are propagated. All '9's is rewritten as '1000...' to prec digits.

RETURNS

_round returns 1 if all '9's rounded up, otherwise zero.

SEE ALSO

_norm

BUGS

No check is made for non-digits in the buffer.

NAME

_stop - end of stack area

SYNOPSIS

```
TEXT *_stop {1};
```

FUNCTION

_stop is checked, as a compile time option, on entry to each function to ensure that the stack pointer will not be reduced below its value. Setting _stop to zero thus effectively disables this check. Normally, if the stack pointer is reduced below the value of _stop, the _memerr condition is raised.

By convention, an initial value of 1 for _stop encourages system specific startup code to reset _stop to a more meaningful value.

SEE ALSO

_memerr

NAME

_tiny - smallest double number

SYNOPSIS

DOUBLE _tiny

FUNCTION

_tiny is the smallest positive representable double number larger than zero.

SEE ALSO

_dzero, _huge

NAME

_unpack - extract fraction from exponent part

SYNOPSIS

```
COUNT _unpack(pd)
    DOUBLE *pd;
```

FUNCTION

_unpack partitions the double at *pd, which should be nonzero, into a fraction in the interval [1/2, 1) times two raised to an integer power, delivers the fraction to *pd and returns the integer power as the value of the function.

RETURNS

_unpack returns the power of two exponent of the double at pd as the value of the function and writes the fraction at *pd. The exponent is generally meaningless if d is zero.

EXAMPLE

```
DOUBLE sqrt(x)
    DOUBLE x;
    {
    COUNT n;

    n = _unpack(&x);
    x = newton(x);
    if (n & 1)
        x =* SQRT2;
    return (_addexp(x, n >> 1));
    }
```

SEE ALSO

_addexp, _frac

NAME

_when - handle exceptions

SYNOPSIS

```
COUNT _when(ptr, c1, c2, ..., cend)
    TEXT **ptr, **c1, **c2, ..., **cend;
```

FUNCTION

_when registers a willingness to handle certain exceptions that may be raised by calls to _raise. The _when/_raise mechanism is used to perform a broad spectrum of stack manipulations normally beyond the scope of the C language, including: Ada exception handling, Pascal nonlocal goto's, Idris process switching, editor interrupt fielding, and math error reporting.

The call to _when causes its argument list and certain non-volatile registers to be left on the stack, where they are made the latest part of a chain of condition handlers. Should a subsequent call to _raise report a condition that is to be handled by this part of the chain, control flow resumes with a return from _when, indicating which condition has been raised. Upon every return, all register variables are restored to their values at the time of the initial call to _when. The _raise call may cause the stack to be cleaned up as part of the return from _when; this is a mandatory prelude to returning from any function that calls _when.

If ptr is not NULL, it is used as the address of a pointer that will be set to point at the latest part of the handler chain; this value may be used by subsequent _raise calls to specify this particular call to _when instead of the normal top of the handler chain. ptr is also used when the stack is cleaned up on return, as the address at which to write the condition being handled.

The conditions c1, c2, etc. each may assume any value except NULL or -1, although there is a strong presumption that the value is a valid data space address of a pointer to a NUL terminated string of characters. A -1 is taken as a cend that indicates no further conditions, while a NULL is taken as a cend that will handle any condition. The leftmost condition argument that will handle a given condition, in the latest part of the handler chain, is chosen to handle the condition.

Since _when plays fast and loose with the stack, it should never be used except as the lone operand in a switch statment, and all _when calls must be carefully coordinated with appropriate _raise calls to stay sane.

RETURNS

_when returns -1 upon return from its initial setup. It returns zero on a cleanup return that reports no condition. Otherwise it returns the ordinal position, within the argument list, of the condition it is handling; a one indicates c1, two means c2, etc. If cend is NULL, its ordinal position will be returned for any condition not otherwise handled.

The stack is cleared, and a non-NULL ptr is used to return the second argument to _raise, if a) either argument to _raise was NULL or b) if a NULL cend is handling the condition.

EXAMPLE

To field interrupts interactively:

```
VOID endup()
    {
    putstr(STDOUT, "?\n", NULL);
    _raise(NULL, NULL);
    }

 ...

FOREVER
    {
    onintr(&endup);
    _when(NULL, NULL);
    if (edit() == EOF)
        exit(YES);
    }
```

SEE ALSO

_raise, enter(II), leave(II)

BUGS

You are not expected to understand this.

Whitesmiths, Ltd.
97 Lowell Road, Concord, Massachusetts 01742
Telephone (617) 369-8499 Telex 951708 SOFTWARE CNCM