III.a. Idris System Interface Library

NAME

Interface - to Idris system

FUNCTION

Programs written in C for operation on the 8086 family under Idris are translated into assembler according to the following specifications:

external identifiers - may be written in both upper and lower case; both cases are significant. The first eight letters must be distinct. Any underscore is left alone. An underscore is prepended to each identifier.

function text - is normally generated into the .text segment and is not to be altered or read as data. External function names are published via .public declarations.

literal data - such as strings, are normally generated into the .data segment. Switch tables are generated into the .text segment.

initialized data - are normally generated into the .data segment. External data names are published via .extern declarations.

uninitialized declarations - result in an .extern declaration as above, one instance per program file. The SMALL model of computation is used, i.e., it is presumed that the segment registers ds and ss contain the same value. The segment register es is never used, nor are segment overrides ever specified by generated code.

function calls - are performed by

1) moving arguments on the stack, right to left. Character data is sign-extended to integer, float is converted to double.

2) calling via "call _func", where _func is assumed by default to be in the same segment,

3) popping the arguments off the stack.

Except for returned value, the registers ax, cx, and dx, the flags, and the floating accumulators are undefined on return from a function call. All other registers are preserved. The returned value is in ax (char sign-extended to integer, integer, pointer to), or in dx/ax (long), or in the primary floating accumulator (float widened to double, double). Note that an intersegment call is generated to any function named by a "-far" flag to p2.86.

floating accumulators - are allocated in the sixteen-byte data area beginning at c_fac, if the 8087 processor is not used; the primary accumulator is at c_fac, the secondary is at c_fac+8. If the 8087 floating point processor is used (-f flag to p2), the primary floating accumulator is fr0 and the secondary is fr7; no other accumulators are modified, and the stack is left balanced at every call and return. The control word is never modified by generated code or runtime support functions; hence it must be set properly at program

startup, and may be modified during program operation. Its value at initialization is probably best for use with the math libraries; it is certainly unwise to specify a precision of less than 53 bits.

stack frames - are maintained by each C function, using bp as a frame pointer. On entry to a function, the call "call c_sav" will stack bp and leave it pointing at the stacked bp, then stack si, di, and bx. Arguments are at 4(bp), 6(bp), etc. and auto storage may be reserved on the stack at -7(bp) on down. To return, the jump "jmp c_ret" will use bp to restore bp, sp, bx, si, and di, then return. The jump "jmp c_fret" will restore the registers in the same way, then do an inter-segment return (as from a function named by a "-far" flag to p2.86). In either case, the previous sp is ignored, so the stack need not be balanced on exit, and none of the potential return registers are used. If within a "near" function it is not necessary to save or restore si, di, and bx, then the sequence "push bp/mov bp,sp" is used on entry and the jump "jmp c_rets" is used to return. If stack checking is requested (-ck flag to p2), the call "call c_savs" will behave as does c_sav, and in addition ensure that adding the quantity in ax to sp will not cause the stack to wrap around nor go below the value in the C variable _stop (__stop in assembler).

data representation - integer is the same as short, two bytes stored less significant byte first. Long integers are stored as two short integers, more significant short first; when copied to dx/ax, the more significant half is in dx. All signed integers are twos complement. Floating numbers are represented as for the 8087 processor, following the IEEE Standard, four bytes for float, eight for double, and are stored in ascending order of significance.

storage bounds - no storage bounds need to be enforced. Two-byte boundaries may be enforced, inside structures and among automatic variables, to significantly enhance performance and to ensure data structure compatibility with machines such as the PDP-11, but boundaries stronger than this are not fully supported by the stacking logic; the compiler may generate incorrect code for passing long or double arguments if a boundary stronger than even is requested.

module name - is not used.

SEE ALSO

c_fac(IV), c_ret(IV), c_rets(IV), c_sav(IV), c_savs(IV)

NAME

Conventions - Idris system subroutines

SYNOPSIS

#include <sys.h>

FUNCTION

All standard system library functions callable from C follow a set of uniform conventions, many of which are supported at compile time by including a standard header file, <sys.h>, at the top of each program. Note that this header is used in addition to the standard header <std.h>. The system header defines various system parameters and a useful macro or two.

Herewith the principal definitions:

```
DIRSIZE - 14, the maximum directory name size
E2BIG - 7, the error codes returned by system calls
EACCES - 13
EAGAIN - 11
EBADF - 9
EBUSY - 16
ECHILD - 10
EDOM - 33
EEXIST - 17
EFAULT - 14
EFBIG - 27
EINTR - 4
EINVAL - 22
EIO -  5
EISDIR - 21
EMFILE - 24
EMLINK - 31
ENFILE - 23
ENODEV - 19
ENOENT - 2
ENOEXEC - 8
ENOMEM - 12
ENOSPC - 28
ENOTBLK - 15
ENOTDIR - 20
ENOTTY - 25
ENXIO - 6
EPERM - 1
EPIPE - 32
ERANGE - 34
EROFS - 30
ESPIPE - 29
ESRCH - 3
ETXTBSY - 26
EXDEV - 18
NAMSIZE - 64, the maximum filename size, counting NUL at end
NSIG - 16, the number of signals, counting signal 0
SIGALRM - 14, the signal numbers
SIGBUS - 10
```

SIGDOM - 7
SIGFPT - 8
SIGHUP - 1
SIGILIN - 4
SIGINT - 2
SIGKILL - 9
SIGPIPE - 13
SIGQUIT - 3
SIGRNG - 6
SIGSEG - 11
SIGSYS - 12
SIGTERM - 15
SIGTRC - 5

The macro isdir(mod) is a boolean rvalue that is true if the mode mod, obtained by a getmod call, is that of a directory. Similarly isblk(mod) tests for block special devices, and ischr(mod) tests for character special devices.

NAME

c86 - compile and link C programs

SYNOPSIS

c86 -[f* o* p* v +*] <files>

FUNCTION

c86 is an instantiation of the generic C driver described in Section II, configured to compile C (with filenames *.c) or as.86 (*.s) source files to object (*.o), and/or to link object files with the standard header and libraries to produce an executable file.

Since a prototype file is a text file, it is easy to vary pathnames or flags for local usage.

SEE ALSO

as.86(II), c(II), pc86, p2.86(II)

NAME

pc86 - compile and link Pascal programs

SYNOPSIS

pc86 -[f* o* p* v +*] <files>

FUNCTION

pc86 is an instantiation of the generic C driver described in Section II, configured to compile Pascal (with filenames *.p), Pascal compatible C (*.c), or as.86 (*.s) source files to object (*.o), and/or to link object files with the standard header and libraries to produce an executable file.

Since a prototype file is a text file, it is easy to vary pathnames or flags for local usage.

SEE ALSO

c(II), c86, ptc(II)

NAME

Crt - C runtime entry

SYNOPSIS

link /lib/Crts.o <main.o>

FUNCTION

All Idris programs begin execution at the start of the .text section; Crts.o is the startup routine that maps an Idris invocation into the standard C call to main.

Idris passes exec arguments on the stack with ac on top, followed by av[0], av[1], etc., whereas main expects a return link on top, followed by ac, then a pointer to av[0]. Similarly, Idris expects a zero return from main (argument to exit) to signal success, whereas a boolean true (non-zero) is returned by C on success. Crts.o makes the necessary changes in both directions.

NAME

Crtp - set up profiling at runtime

SYNOPSIS

link /lib/Crtp.o /lib/Crts.o <main.o> /lib/libi.*

FUNCTION

Crtp.o is the startup routine that enables profiling to occur, by calling the portable C function _profil(), which sets up profiling buffer areas and requests the operating system to begin periodically recording the user PC location. Crtp.o also contains the function entry counting routine called by properly instrumented functions (compiled with the p2 option "-p"). The one-byte flag _penable, if non-zero, enables this routine to perform counting. Otherwise, entry counting is disabled.

Finally, Crtp.o contains the parameters controlling how profiling is performed. These are stored as a standard profile file header, whose start is marked by the symbol _pheader. The two parameters most likely to be modified are the number of function entry counters to be maintained, a short int at _pheader+2, and the number of bytes of text that are to correspond to each element of the PC histogram, the fourth int counting from _pheader+4. Note that both of these are modified before being output in the profile header, where the first becomes the number of bytes occupied by entry counters, and the second becomes the binary fraction corresponding to the integer scaling factor originally given. By default, Crtp.o provides 100 function entry counters and a resolution of 8 text bytes per histogram entry.

Crtp.o must be the first module in the .text section of a program, and so must appear first on the link command line.

EXAMPLE

To set up 256 function entry counters, and 4 bytes of text per histogram entry, for a PDP-11 executable file:

```
% db11 -u prog11
prog11: 11400T + 1340D + 0B
__pheader+2 ps
__pheader+2     100
u
256
.
__pheader+10 ps
__pheader+10          8
u
4
.
q
```

Or to set up 400 entry counters and a scaling factor of 2 bytes per histogram entry, for a MC68000 executable file:

```
% db68k -u prog68k
prog68k: 13542T + 1544D + 0B
```

```
__pheader+2 ps
__pheader+2     100
u
400
.
__pheader+16 pl
__pheader+16              8
u
2
.
q
```

SEE ALSO

The profile file format description is in Section III of the Idris Programmers' Manual. The portable profiling setup functions are described in Section IV of the Idris Programmers' Manual, and the machine-dependent function entry counting routine is described in Section IV of the current manual. The profile post-processor prof is described in Section II of this manual.

BUGS

Because of the original UNIX V6 specification for the histogram scaling factor (retained here), a factor of 2 bytes of text per histogram entry is the smallest that can be specified.

NAME

_pname - program name

SYNOPSIS

```
TEXT *pname;
```

FUNCTION

_pname is the (NUL terminated) name by which the program was invoked, as obtained from the command line argument zero. It overrides any name supplied by the program at compile time.

It is used primarily for labelling diagnostic printouts.

NAME

close - close a file

SYNOPSIS

```
ERROR close(fd)
    FILE fd;
```

FUNCTION

close closes the file associated with the file descriptor fd, making fd available for future open or create calls.

RETURNS

close returns zero, if successful, or a negative number, which is the Idris error return code, negated.

EXAMPLE

To copy an arbitrary number of files:

```
while (0 < ac && 0 <= (fd = open(av[--ac], READ, 0)))
    {
    while (0 < (n = read(fd, buf, BUFSIZE)))
        write(STDOUT, buf, n);
    close(fd);
    }
```

SEE ALSO

create, open, remove, uname

NAME

create - open an empty instance of a file

SYNOPSIS

```
FILE create(fname, mode, rsize)
    TEXT *fname;
    COUNT mode;
    BYTES rsize;
```

FUNCTION

create makes a new file with name fname, if it did not previously exist, or truncates the existing file to zero length. An existing file has its permissions left alone; otherwise if the filename returned by uname is a prefix of fname, the (newly created) file is given restricted access (0600); if not, the file is given general access (0666). If (mode == 0) the file is opened for reading, else if (mode == 1) it is opened for writing, else (mode == 2) of necessity and the file is opened for updating (reading and writing).

rsize is the record size in bytes, which must be nonzero on many systems if the file is not to be interpreted as ASCII text. It is ignored by Idris, but should be present for portability.

RETURNS

create returns a file descriptor for the created file or a negative number, which is the Idris error return code, negated.

EXAMPLE

```
if ((fd = create("xeq", WRITE, 1)) < 0)
    putstr(STDERR, "can't create xeq\n", NULL);
```

SEE ALSO

close, open, remove, uname

NAME
exit - terminate program execution

SYNOPSIS
```
VOID exit(success)
    BOOL success;
```

FUNCTION
exit calls all functions registered with onexit, then terminates program execution. If success is non-zero (YES), a zero byte is returned to the invoker, which is the normal Idris convention for successful termination. If success is zero (NO), a one is returned to the invoker.

RETURNS
exit will never return to the caller.

EXAMPLE
```
if ((fd = open("file", READ)) < 0)
    {
    putstr(STDERR, "can't open file\n", NULL);
    exit(NO);
    }
```

SEE ALSO
onexit

NAME

lseek - set file read/write pointer

SYNOPSIS

```
COUNT lseek(fd, offset, sense)
    FILE fd;
    LONG offset;
    COUNT sense;
```

FUNCTION

lseek uses the long offset provided to modify the read/write pointer for the file fd, under control of sense. If (sense == 0) the pointer is set to offset, which should be positive; if (sense == 1) the offset is algebraically added to the current pointer; otherwise (sense == 2) of necessity and the offset is algebraically added to the length of the file in bytes to obtain the new pointer. Idris uses only the low order 24 bits of the offset; the rest are ignored.

The call lseek(fd, 0L, 1) is guaranteed to leave the file pointer unmodified and, more important, to succeed only if lseek calls are both acceptable and meaningful for the fd specified. Other lseek calls may appear to succeed, but without effect, as when rewinding a terminal.

RETURNS

lseek returns the file descriptor if successful, or a negative number, which is the Idris error return code, negated.

EXAMPLE

To read a 512-byte block:

```
BOOL getblock(buf, blkno)
    TEXT *buf;
    BYTES blkno;
    {
    lseek(STDIN, (LONG) blkno << 9, 0);
    return (read(STDIN, buf, 512) != 512);
    }
```

NAME
onexit - call function on program exit

SYNOPSIS
```
VOID (*onexit())(pfn)
    VOID (*(*pfn)())();
```

FUNCTION
onexit registers the function pointed at by pfn, to be called on program exit. The function at pfn is obliged to return the pointer returned by the onexit call, so that any previously registered functions can also be called.

RETURNS
onexit returns a pointer to another function; it is guaranteed not to be NULL.

EXAMPLE
```
IMPORT VOID (*(*nextguy)())(), (*thisguy())();

if (!nextguy)
    nextguy = onexit(&thisguy);
```

SEE ALSO
exit, onintr

BUGS
The type declarations defy description, and are still wrong.

NAME

onintr - capture interrupts

SYNOPSIS

```
VOID onintr(pfn)
    VOID (*pfn)();
```

FUNCTION

onintr ensures that the function at pfn is called on a broken pipe, or on the occurrence of an interrupt (DEL key) or hangup generated from the keyboard of a controlling terminal. Any earlier call to onintr is overridden.

The function is called with one integer argument, whose value is always zero, and must not return; if it does, a message is output to STDERR and an immediate error exit is taken.

If (pfn == NULL) then these interrupts are disabled (turned off). Any disabled interrupts are not, however, turned on by a subsequent call with pfn not NULL.

RETURNS

Nothing.

EXAMPLE

A common use of onintr is to ensure a graceful exit on early termination:

```
    onexit(&rmtemp);
    onintr(&exit);
    ...
VOID rmtemp()
    {
    remove(uname());
    }
```

Still another use is to provide a way of terminating long printouts, as in an interactive editor:

```
    while (!enter(docmd, NULL))
        putstr(STDOUT, "?\n", NULL);
    ...
VOID docmd()
    {
    onintr(&leave);
```

SEE ALSO

onexit

NAME

open - open a file

SYNOPSIS

```
FILE open(fname, mode, rsize)
    TEXT *fname;
    COUNT mode;
    BYTES rsize;
```

FUNCTION

open opens a file with name fname and assigns a file descriptor to it. If (mode == 0) the file is opened for reading, else if (mode == 1) it is opened for writing, else (mode == 2) of necessity and the file is opened for updating (reading and writing).

rsize is the record size in bytes, which must be nonzero on many systems if the file is not to be treated as ASCII text. It is ignored by Idris, but should be present for portability.

RETURNS

open returns a file descriptor for the opened file or a negative number, which is the Idris error return code, negated.

EXAMPLE

```
if ((fd = open("xeq", WRITE, 1)) < 0)
    putstr(STDERR, "can't open xeq\n", NULL);
```

SEE ALSO

close, create

NAME

read - read from a file

SYNOPSIS

```
COUNT read(fd, buf, size)
    FILE fd;
    TEXT *buf;
    BYTES size;
```

FUNCTION

read reads up to size characters from the file specified by fd into the buffer starting at buf.

RETURNS

If an error occurs, read returns a negative number which is the Idris error code, negated; if end of file is encountered, read returns zero; otherwise the value returned is between 1 and size, inclusive. When reading from a disk file, size bytes are read whenever possible.

EXAMPLE

To copy a file:

```
while (0 < (n = read(STDIN, buf, BUFSIZE)))
    write(STDOUT, buf, n);
```

SEE ALSO

write

NAME

remove - remove a file

SYNOPSIS

```
FILE remove(fname)
    TEXT *fname;
```

FUNCTION

remove deletes the file fname from the Idris directory structure. If no other names link to the file, the file is destroyed. If the file is opened for any reason, however, destruction will be postponed until the last close on the file.

If the file is a directory, remove will not attempt to remove it.

RETURNS

remove returns zero, if successful, or a negative number, which is the Idris error return code, negated.

EXAMPLE

```
if (remove("temp.c") < 0)
    putstr(STDERR, "can't remove temp file\n", NULL);
```

SEE ALSO

create

NAME

sbreak - set system break

SYNOPSIS

```
TEXT *sbreak(size)
    BYTES size;
```

FUNCTION

sbreak moves the system break, at the top of the data area, algebraically up by size bytes, rounded up as necessary to placate memory management hardware.

RETURNS

If successful, sbreak returns a pointer to the start of the added data area; otherwise the value returned is NULL.

EXAMPLE

```
if (!(p = sbreak(nsyms * sizeof (symbol))))
    {
    putstr(STDERR, "not enough room!\n", NULL);
    exit(NO);
    }
```

NAME

uname - create a unique file name

SYNOPSIS

```
TEXT *uname()
```

FUNCTION

uname returns a pointer to the start of a NUL terminated name which is guaranteed not to conflict with normal user filenames. The name is, in fact, unique to each Idris process, and may be modified by a suffix, so that a family of process-unique files may be dealt with. The name may be used as the first argument to a create, or subsequent open, call, so long as any such files created are removed before program termination. It is considered bad manners to leave scratch files lying about.

RETURNS

uname returns the same pointer on every call during a given program invocation. It takes the form "/tmp/t#####" where ##### is the processid in octal. The pointer will never be NULL.

EXAMPLE

```
if ((fd = create(uname(), WRITE, 1)) < 0)
    putstr(STDERR, "can't create sort temp\n", NULL);
```

SEE ALSO

close, create, open, remove

BUGS

A program invoked by the exec system call, without a fork, inherits the Idris processid used to generate unique names. Collisions can occur if files so named are not meticulously removed.

NAME

write - write to a file

SYNOPSIS

```
COUNT write(fd, buf, size)
    FILE fd;
    TEXT *buf;
    BYTES size;
```

FUNCTION

write writes size characters starting at buf to the file specified by fd.

RETURNS

If an error occurs, write returns a negative number which is the Idris error code, negated; otherwise the value returned should be size.

EXAMPLE

To copy a file:

```
while (0 < (n = read(STDIN, buf, size)))
    write(STDOUT, buf, n);
```

SEE ALSO

read

NAME

xecl - execute a file with argument list

SYNOPSIS

```
COUNT xecl(fname, sin, sout, flags, s0, s1, ..., NULL)
    TEXT *fname;
    FILE sin, sout;
    COUNT flags;
    TEXT *s0, *s1, ...
```

FUNCTION

xecl invokes the program file fname, connecting its STDIN to sin and STDOUT to sout and passing it the string arguments s0, s1, ... If (!(flags & 3)) fname is invoked as a new process; xecl will wait until the command has completed and will return its status to the calling program. If (flags & 1) fname is invoked as a new process and xecl will not wait, but will return the processid of the child. If (flags & 2) fname is invoked in place of the current process, whose image is forever gone. In this case, xecl will never return to the caller.

To the value of flags may be added a 4 if the processing of interrupt and quit signals for fname is to revert to system handling. The value of flags may also be incremented by 8 if the effective userid is to be made the real userid before fname is executed. If sin is not equal to STDIN, or if sout is not equal to STDOUT, the file (sin or sout) is closed before xecl returns.

If fname does not contain a '/', then xecl will search an arbitrary series of directories for the file specified, by prepending to fname each path specified by the global variable _paths before trying to execute it. _paths is of type pointer to TEXT, and points to a NUL terminated series of directory paths separated by '|'s.

If the file eventually found has execute permission, but is not in executable format, /bin/sh is invoked with the current prefixed version of fname as its first argument and, following fname, an argument vector composed of s0, s1, ...

RETURNS

If fname cannot be invoked, xecl will fail. If (!(flags & 3)) xecl returns YES if the command executed successfully, otherwise NO; if (flags & 1) xecl returns the id of the child process, if one exists, otherwise zero; if (flags & 2) xecl will never return to the caller.

In all cases, if fname cannot be executed, an appropriate error message is written to STDERR.

EXAMPLE

```
if (!xecl(pgm, STDIN, create(file, WRITE), 0, f1, f2, NULL))
    putstr(STDERR, pgm, " failed\n", NULL);
```

SEE ALSO

xecv

NAME

xecv - execute a file with argument vector

SYNOPSIS

```
COUNT xecv(fname, sin, sout, flags, av)
    TEXT *fname;
    FILE sin, sout;
    COUNT flags;
    TEXT **av;
```

FUNCTION

xecv invokes the program file fname, connecting its STDIN to sin and STDOUT to sout and passing it the string arguments specified in the NULL terminated vector av. Its behavior is otherwise identical to xecl.

SEE ALSO

xecl