III.c. UDI System Interface Library

NAME

Interface - to UDI systems

FUNCTION

Programs written in C for operation on the 8086 family under ISIS, RMX-86, or RMX-88 use the Intel 8086 assembler and the standard UDI interface. They are translated into assembler according to the following specifications:

external identifiers - may be written in both upper and lower case, but only one case is 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 code 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 code segment.

initialized data - are normally generated into the data segment. External data names are published via extrn declarations, with the attribute near, from within the data segment.

uninitialized declarations - result in an extrn 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 "near", i.e., 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 st(0) and the secondary is st(7); 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" may be used instead. 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 taken as the first defined external encountered and is published via a name declaration.

SEE ALSO

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

NAME

Conventions - UDI system subroutines

SYNOPSIS

```
#include <udi.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, <udi.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, primarily of use purely among the UDI interface functions. This file can safely be ignored by most programmers, but is documented here for completeness.

Herewith the definitions:

FAIL - the failure code returned by many functions
ISBIN - the bit in the field open of a WCB indicating binary I/O
ISOPEN - the bit in the field open of a WCB indicating file is open
MCREATE - internal code for a create operation
MOPEN - internal code for an open operation
MREMOVE - internal code for a remove operation
WCB - a control structure used to map file descriptors to UDI connections

NAME

c - compiling C programs

SYNOPSIS

```
submit c(sfile)
```

FUNCTION

c is a command sequence definition file that causes a C source file to be compiled and assembled. It does so by invoking the compiler passes and asm86 in the proper order, then deleting the intermediate files.

The C source file is at sfile.c, where sfile is the name contained in parentheses on the submit command line. The object file output from asm86 is put at sfile.obj.

EXAMPLE

To compile test.c:

```
    -submit c(test)
```

SEE ALSO

clink, pc

NAME

pc - compiling Pascal programs

SYNOPSIS

submit pc(sfile)

FUNCTION

pc is a command sequence definition file that causes a Pascal source file to be compiled and assembled. It does so by invoking the compiler passes and asm86 in the proper order, then deleting the intermediate files.

The Pascal source file is at sfile.p, where sfile is the name contained in parentheses on the submit command line. The object file output from asm86 is put at sfile.obj.

EXAMPLE

To compile test.p:

```
    -submit pc(test)
```

SEE ALSO

c, clink

NAME

clink - linking C programs

SYNOPSIS

```
submit clink('<files.obj>', <exename>)
```

FUNCTION

Programs written in C for operation under UDI must be linked with certain object modules that implement the standard C runtime environment. clink is a command sequence definition file that combines the list of object files '<files.obj>' with a header file and modules selected from the portable C library to form an executable image at <exename>.

The header calls the function _main(), described in this section, which parses a command line, redirects standard input and output as necessary, calls the user provided main(ac, av), and passes the value returned by main() on to exit(). All of this malarkey can be circumvented if <files.obj> contains a defining instance of _main().

<files.obj> can be one or more object modules produced by the C compiler, or produced from asm86 source that satisfies the interface requirements of C, as described under Interface in this section. The important thing is that the function main(), or _main(), be defined somewhere among these files, along with any other modules needed and not provided by the standard C library.

EXAMPLE

To compile and run the program echo.c:

```
-submit c(echo)
-submit clink(echo.obj, echo)
-run echo hello world
hello world
```

SEE ALSO

c, pc

NAME

chdr - C runtime entry

FUNCTION

All programs using the UDI begin execution at the start of chdr.obj, which is typically linked first when constructing an executable file. fpphdr.obj precedes chdr.obj if the 8087 floating point processor must be initialized at the start of the program. chdr.obj sets up the segment registers for the "small" memory model, sets the stack pointer to the top of the space reserved for the data segment, then calls _main. If the stack top fence _stop has not been otherwise initialized, the startup code also sets _stop to point just beyond the end of initialized data.

SEE ALSO

_main

NAME

_main - setup for main call

SYNOPSIS

```
BOOL _main()
```

FUNCTION

_main is the function called whenever a C program is started. It opens STDIN, STDOUT, and STDERR for console input/output, obtains the command line, parses it into argument strings, redirects STDIN and STDOUT as specified in the command line, then calls main().

The command line is interpreted as a series of strings separated by whitespace, or beginning with ´<´ or ´>´. If a string begins with a ´<´, the remainder of the string is taken as the name of a file to be opened for reading text and used as the standard input, STDIN. If a string begins with a ´>´, the remainder of the string is taken as the name of a file to be created for writing text and used as the standard output, STDOUT. All other strings are taken as argument strings to be passed to main. The command name, av[0], is written into _pname.

EXAMPLE

To avoid the loading of _main and all of the file I/O code it calls on, one can provide a substitute _main instead:

```
BOOL _main()
    {
    <program body>
    }
```

RETURNS

_main returns the boolean value obtained from the main call, which is then passed to exit.

SEE ALSO

_pname, exit

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.

SEE ALSO

_main

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 UDI 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. 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).

If (rsize == 0), carriage returns are deleted on input, and a carriage return is injected before each newline on output. If (rsize != 0), data is transmitted unaltered.

RETURNS

create returns a file descriptor for the created file or a negative number, which is the UDI 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. success is ignored by UDI.

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. The number of bits of loff which are used is operating system dependent.

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.

RETURNS

lseek returns the file descriptor if successful, or a negative number, which is the UDI 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 is supposed to ensure that the function at pfn is called on the occurrence of an interrupt generated from the keyboard of a controlling terminal. (Typing a delete DEL, or sometimes a ctl-C ETX, performs this service on many systems.)

onintr is currently a dummy, on this system, so pfn is never called.

RETURNS

Nothing.

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).

If (rsize == 0), carriage returns are deleted on input, and a carriage return is injected befor each newline on output. If (rsize != 0), data is transmitted unaltered.

RETURNS

open returns a file descriptor for the created file or a negative number, which is the UDI 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. If the file was created or opened with (rsize == 0), or if STDIN is not redirected from the console, carriage returns are discarded on input.

RETURNS

If an error occurs, read returns a negative number which is the UDI error code, negated; if end of file is encountered, read returns zero; otherwise the value returned is between 1 and size, inclusive.

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 operating system directory structure.

RETURNS

remove returns zero, if successful, or a negative number, which is the UDI 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 reserves size bytes in an area above the defined data area and below the stack.

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 may be modified by a suffix of up to three letters, 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, which is currently the string "ctempc.". 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

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. If the file was created or opened with (rsize == 0), or if STDOUT or STDERR is not redirected from the console, each newline output is preceded by a carriage return.

RETURNS

If an error occurs, write returns a negative number which is the UDI 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