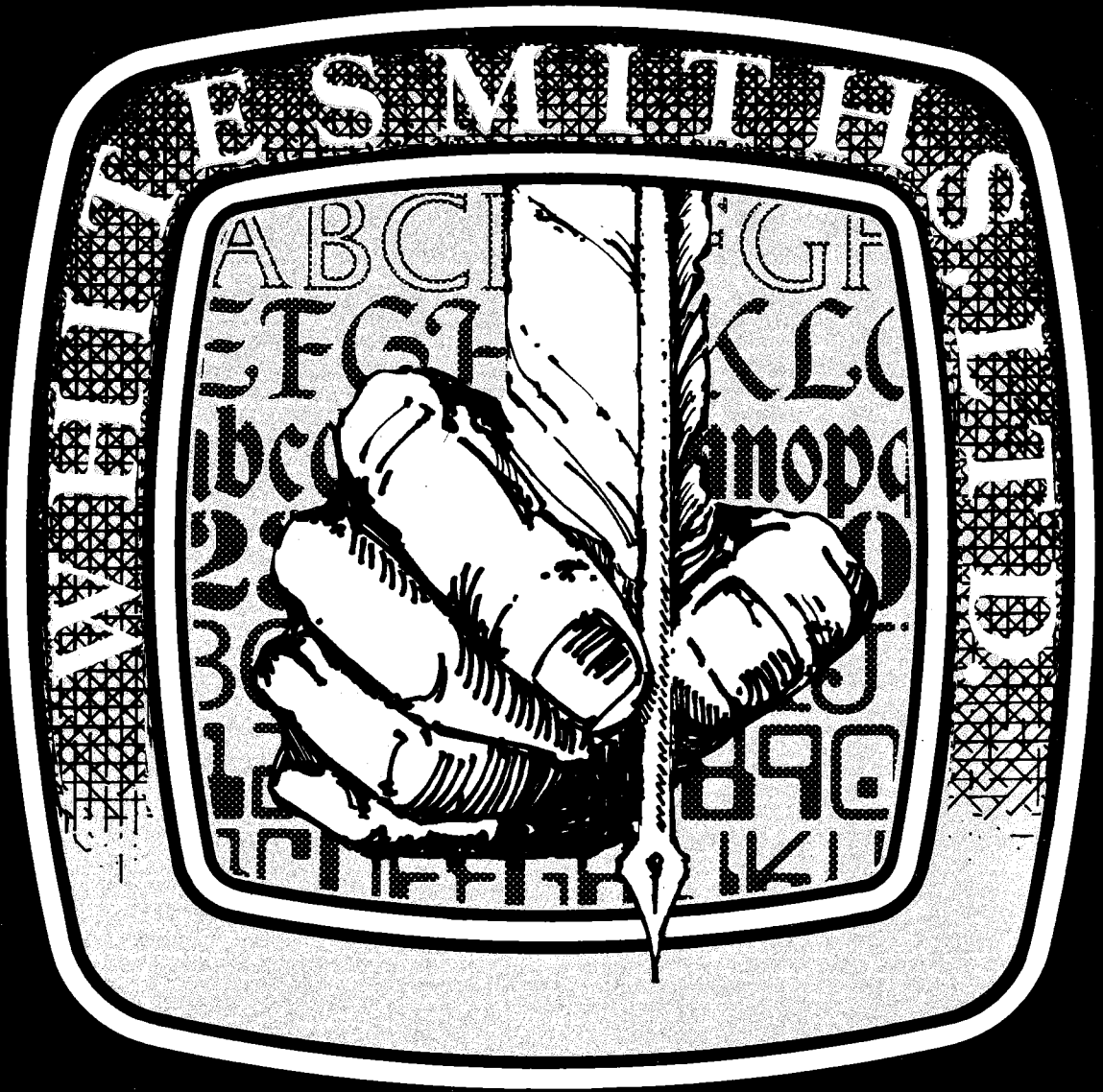
Whitesmiths, Ltd.
C Interface Manual
for MC68000



COMPUTER TOOLS INTERNATIONAL
649 STRANDER BLVD. SUITE E
SEATTLE, WA 98188
(206) 575-3606

Whitesmiths, Ltd.

C INTERFACE MANUAL FOR MC68000

Date: March 1983

The C language was developed at Bell Laboratories by Dennis Ritchie; Whitesmiths, Ltd. has endeavored to remain as faithful as possible to his language specification. The external specifications of the Idris operating system, and of most of its utilities, are based heavily on those of UNIX, which was also developed at Bell Laboratories by Dennis Ritchie and Ken Thompson. Whitesmiths, Ltd. gratefully acknowledges the parentage of many of the concepts we have commercialized, and we thank Western Electric Co. for waiving patent licensing fees for use of the UNIX protection mechanism.

The successful implementation of Whitesmiths' compilers, operating systems, and utilities, however, is entirely the work of our programming staff and allied consultants.

For the record, UNIX is a trademark of Bell Laboratories; IAS, RSTS/E, VAX, VMS, P/OS, PDP-11, RT-11, RSX-11M, and nearly every other term with an 11 in it all are trademarks of Digital Equipment Corporation; CP/M is a trademark of Digital Research Co.; MC68000 and VERSAdos are trademarks of Motorola Inc.; ISIS and iRMX are trademarks of Intel Corporation; A-Natural, Idris, and ctext are trademarks of Whitesmiths, Ltd. C is not.

C INTERFACE MANUAL FOR MC68000

SECTIONS

SCOPE

This manual describes the MC68000 dependent aspects of the C programming environment provided by Whitesmiths, Ltd. In addition, it documents all of the utilities necessary for building new programs. Section I introduces the conventions and describes the format used by the Idris assembler. Section II succinctly describes the programming utilities of Idris, which also serve as cross support utilities for other host machines. Each subsection of Section III describes the library functions that interface the portable C library to Idris, VERSAdos, or CP/M-68k. Section IV describes the runtime routines called upon by code produced by the MC68000 C compiler.

Information on the C language and the portable library may be found in the C Programmers' Manual, while information peculiar to other machines supported by Whitesmiths, Ltd. is given in other C Interface Manuals.

THIS MANUAL IS PROVIDED WITH SEVERAL SOFTWARE PACKAGES, SOME OF WHICH USE ONLY A SUBSET OF THE FACILITIES DOCUMENTED. THE PRESENCE OF A MANUAL PAGE HERE DOES NOT IMPLY THAT THE CORRESPONDING SOFTWARE IS ALSO SUPPLIED.

TABLE OF CONTENTS

I. Idris Assembler Conventions for MC68000

II. Programming Utilities

III.a. Idris System Interface Library

III.b. VERSAdos System Interface Library

III.c. CP/M-68k System Interface Library

IV. Machine Support Library for MC68000

I. Idris Assembler Conventions for MC68000

NAME

As.68k - The MC68000 Assembly Language

FUNCTION

The as.68k language facilitates writing machine level code for the MC68000 family of computers. Its primary design goal is to express the readable output of the C compiler code generator p2.68k in a form that can be translated as rapidly as possible to relocatable object form; but it also includes limited facilities for aiding the hand construction of machine code. What it does not include are listing output control, macro definitions, conditional assembly directives, or branch shortening logic.

SYNTAX

Source code is taken as a sequence of zero or more lines, where each line is a sequence of no more than 511 characters terminated by either a newline or a semicolon. The asterisk '*' is a comment delimiter; all characters between the asterisk and the next newline are equivalent to a single newline. Each line may contain at most one command, which is specified by a sequence of zero or more tokens; an empty line is the null command, which does nothing.

A token is any one of the characters "()!+-,:=", the sequence "-(", the sequence ")+", a number, or an identifier. A number is the ASCII value of a character x when written as 'x, or a digit string. A character x may be replaced by the escape sequence '\c, where c is in the sequence "btnvfr" (in either case) for the octal values [010, 015], or where c is one to three decimal digits taken as an octal number giving the value of the character. A digit string is a decimal digit, followed by zero or more letters and digits. If the leading characters are "0x" or "0X", the constant is a hexadecimal number and may contain the letters 'a' through 'f', in either case, to represent the digit values 10 through 15, respectively. Otherwise a leading '0' implies an octal number, which nevertheless may contain the digits '8' and '9'. A non-zero leading digit implies a decimal number.

IDENTIFIERS

An identifier is one or more letters and digits, beginning with a letter; letters are characters in the set [a-zA-Z._~], with uppercase distinguished from lowercase; at most eleven characters of an identifier are retained. Tokens that might be otherwise confused must be separated by whitespace, which consists of the ASCII space ' ' and all non-printing characters.

There are many pre-defined identifiers, most of which represent machine instructions. New identifiers are defined either in a label prefix or in a defining command; the scope of a definition is from its defining occurrence through the end of the source text. A label prefix consists of an identifier followed by a colon ':'; any number of labels may occur at the beginning of a command line. A label prefix defines the identifier as being equal to the (relocatable) address of the next byte for which code

is to be generated. A defining command consists of an identifier, followed by an equals '=', followed by an expression; it defines the identifier as being equal to the value of the expression, which must contain no undefined identifiers.

There are also twenty numeric labels, identified by a digit sequence whose value is in the range [0, 9], followed immediately by a 'b' or an 'f'. A definition of the form "#:" or "# = expr", where # is in the range [0, 9] makes the corresponding #b numeric label defined and renders the corresponding #f undefined; further, any previous references to #f are given this definition. Thus, #b always refers to the last definition of #, which must exist, and #f always refers to the next definition, which must eventually be defined.

EXPRESSIONS

Expressions consist of an alternation of terms and operators. A term is a number, a numeric label, or an identifier. There are three operators: plus "+" for addition, minus "-" for subtraction, and not "!" for bitwise equivalence. If an expression begins with an operator, an implicit zero term is prepended to the expression; if an expression ends with an operator, an error is reported. Thus plus, minus, and not have their usual unary meanings.

Two terms may be added if at most one is undefined or relocatable; two terms may be subtracted if the right is absolute, relocatable to the same base as the left, or involves the same undefined symbol as the left; two terms may be equivalenced only if both are absolute. The base of each term is examined only when an expression is first encountered, so that symbols named as forward references are always considered undefined in this context.

RELOCATION

Relocation is always in terms of one of three code sections: the text section, which normally receives all executable instructions and is normally read only; the data section, which normally receives all initialized data areas and is normally read/write; and the bss section, which can accept only space reservations and is initialized to zeros at start of execution. The pre-defined variable dot "." is maintained as the location counter; its value is the (relocatable) address of the next byte for which code is to be generated at the start of each command line.

The commands ".text", ".data", and ".bss" switch between sections; each sets dot to wherever code generation left off for that section and forces an even byte boundary. Initially, all three sections are of length zero and dot is in the text section. Space may be reserved by a definition of the form ". = . + #", where # is an absolute expression whose value advances the location counter. One byte of space may be conditionally reserved by the command ".even", which ensures that dot is henceforth on an even byte boundary, relative to the start of the current section.

GLOBALS

The command ".globl ident [, ident]*" declares the one or more identifiers to be globally known, i.e., available for reference by other modules. Undefined identifiers may thus be given definitions by other modules at load time, provided there is exactly one globally known defining instance for each such identifier.

A special form of global reference is given by the command ".comm ident, #", which makes ident globally known and treats it as undefined in the current module; if no defining instance exists at link time, however, then this command requests that at least # bytes be reserved in the combined bss section and that the identifier be defined as the address of the first byte of that area (thus bss for "Block Started by Symbol"). As before, # represents an expression whose value is absolute.

CODE GENERATION

Code may be generated into the current section, provided it is text or data, a byte, word, or longword at a time, by one of the commands ".byte # [, #]*", ".word # [, #]*", or ".long # [, #]*". For byte and word generation, each # is an absolute expression whose least significant byte or word defines the next code to be generated; longword generation may also involve relocatable expressions. A word or longword must begin on an even byte boundary.

To simplify generating ASCII strings, an arbitrary string within double quotes, such as "string", may be used as a shorthand for, in this case, ".byte 's, 't, 'r, 'i, 'n, 'g". The usual escape sequences for characters apply.

ADDRESS MODES

All remaining commands are machine instructions. Each is introduced by an identifier defined as an instruction and, depending upon the format defined for that instruction, is followed by zero, one, or two address modes. Some address modes are constrained to be absolute expressions, others must be relocatable addresses within some narrow range of the current location counter, others must be certain machine registers, still others are relatively general.

General address modes are built from expressions and the pre-defined register identifiers "d0" through "d7", "a0" through "a7", "d0.l" through "d7.l", "a0.l" through "a7.l", "d0.w" through "d7.w", "a0.w" through "a7.w", "sp", "sp.l", "sp.w", "pc", "sr", and "ccr". The ".l" (for longword) and ".w" (for word) modifiers are only significant when used with second index registers; "sp" and its variants are synonyms for "a7"; "pc" is the program counter designator for program counter with index mode, and "sr" and "ccr" both designate the status register.

A register address mode consists of simply a register expression. Autoincrement is indicated by the form "(reg)+", autodecrement by "-(reg)", in-

dexing by "expr(reg)", absolute or program counter relative by "expr", and immediate by "#expr", where expr is any expression and reg is any A register expression. The form "expr" becomes program counter relative if it refers to a defined symbol within reach in the same section as the instruction, or if the flag -s was specified to the translator and the symbol is undefined; a non-relocatable expression is made either absolute short or absolute long as needed; an undefined symbol is made absolute if program counter relative cannot be used. In this last case, the mode absolute short is used if -a was specified to the assembler, and absolute long otherwise.

The other index modes are indicated by "expr(pc, reg)" or "expr(reg1, reg2)", where expr is an absolute expression that can be represented within a signed byte, reg and reg2 are any short or long registers, and reg1 is any A register.

Another form of address mode is a mask, for use with the movem instruction. A mask may consist of any A or D register alone, a range of A or D registers such as "d0-d3", or a union of masks such as "a2/d0-d3". The complement of a mask may be obtained by writing "d3-d5 ! 0", which selects all registers except d3, d4, and d5.

PRE-DEFINED IDENTIFIERS

Pre-defined identifiers fall into three groups: registers, command keywords, and instructions. The registers are listed above; other names for registers may only be formed by definitions of the form "temp = d3". The keywords, and the command formats they imply are:

```
.bss
.byte   # [, #]*
.comm   ident, #
.data
.even
.globl  ident [, ident]*
.long   # [, #]*
.text
.word   # [, #]*
```

where # implies an expression, ident is an identifier, and "[x]*" implies zero or more repetitions of x.

To complete the list of commands other than instructions:

```
ident = expr     / defines ident as expr
"string"         / generates code for string characters
```

Instructions frequently have several alternate encodings, depending upon the combination of operands used. Hence, only the names pre-defined in as.68k are listed here. They are:

abcd	ble.s	dbhi	movep.w	sbcd
add.b	bls	dble	moveq	scc
add.l	bls.s	dbls	moveq.l	scs
add.w	blt	dblt	mtccr	seq
adda.l	blt.s	dbmi	mtsr	sf
adda.w	bmi	dbne	mtusp	sge
addi.b	bmi.s	dbpl	muls	sgt
addi.l	bne	dbra	mulu	shi
addi.w	bne.s	dbt	nbcd	sle
addq.b	bpl	dbvc	neg.b	sls
addq.l	bpl.s	dbvs	neg.l	slt
addq.w	bra	divs	neg.w	smi
addx.b	bra.s	divu	negx.b	sne
addx.l	bset	eor.b	negx.l	spl
addx.w	bsr	eor.l	negx.w	st
and.b	bsr.s	eor.w	nop	stop
and.l	btst	eori.b	not.b	sub.b
and.w	bvc	eori.l	not.l	sub.l
andi.b	bvc.s	eori.w	not.w	sub.w
andi.l	bvs	exg	or.b	suba.l
andi.w	bvs.s	ext.l	or.l	suba.w
asl.b	chk	ext.w	or.w	subi.b
asl.l	clr.b	jmp	ori.b	subi.l
asl.w	clr.l	jsr	ori.l	subi.w
asr.b	clr.w	lea	ori.w	subq.b
asr.l	cmp.b	link	pea	subq.l
asr.w	cmp.l	lsl.b	reset	subq.w
bcc	cmp.w	lsl.l	rol.b	subx.b
bcc.s	cmpa.l	lsl.w	rol.l	subx.l
bchg	cmpa.w	lsr.b	rol.w	subx.w
bclr	cmpi.b	lsr.l	ror.b	svc
bcs	cmpi.l	lsr.w	ror.l	svs
bcs.s	cmpi.w	mfsr	ror.w	swap
beq	cmpm.b	mfusp	roxl.b	tas
beq.s	cmpm.l	move.b	roxl.l	trap
bge	cmpm.w	move.l	roxl.w	trapv
bge.s	dbcc	move.w	roxr.b	tst.b
bgt	dbcs	movea.l	roxr.l	tst.l
bgt.s	dbeq	movea.w	roxr.w	tst.w
bhi	dbf	movem.l	rte	unlk
bhi.s	dbge	movem.w	rtr	
ble	dbgt	movep.l	rts	

Note that special mnemonics "mfusp" and "mtusp" are used to access the user stack pointer, rather than a "move" instruction with a special register name. Special mnemonics "mfsr", "mtsr", and "mtccr" are included for the operations move from/to status register and move to condition codes. All five of these mnemonics are followed by a single effective address giving the second operand for the move. Also, note that as.68k requires the size field on instructions that can operate on different sized data; there are no default sizes.

as.68k is not guaranteed to pick the optimal instruction where there is a choice. For example, add.l #2,d0 will not select the addq.l operation. In general, it is wise always to state exactly what you mean.

ERROR MESSAGES

Errors are reported by the as.68k translator in English, albeit terse, as opposed to the conventional assembler flags. Any message ending in an exclamation mark '!' indicates problems with the translator per se (they should "never happen") and hence should be reported to the maintainers. Here is a complete list of errors that can be produced by erroneous input or by an inadequate operating environment:

```
#b undefined                    data register required
.comm defined                   illegal .=
A reg or PC required            illegal character
absolute expr required          missing / operand
address register required       missing immediate
bad #:                          missing term
bad #f or #b                    missing xxx
bad .comm size                  redefinition of xxx
bad command                     register required
bad index or indirect           reloc + reloc
bad mask range                  relocatable byte
bad movem mask                  relocatable word
bad operand(s)                  string too large
bad temp file read              too many symbols
byte pc range                   undefined #f
can't backup .                  unknown instruction
can't create temp file          word on odd boundary
can't create xxx                word pc range
can't load .bss                 x ! reloc
can't read xxx                  x - reloc
can't write object file
```

In all cases, "xxx" stands for an actual character, identifier, or filename supplied by as.68k.

II. Programming Utilities

NAME

Introduction - the programming utilities

FUNCTION

The utilities described in this section are provided with the Idris operating system, or with a cross-compiler on any other operating system, to build and debug programs written in assembler, C, or Pascal. Since modules may be separately translated to object code, then combined (by link) in one or more stages to make an executable entity, each module can be written in the most appropriate of the languages supported.

Libraries may be constructed (by lib) from modules that are included only as needed (by link) in building a program. Thus, the library routines provided impose no space penalty for programs that don't use them; and each user is at liberty to add to this set or alter any of its members.

Unless explicitly labelled as machine dependent, every utility in this section can be used to develop programs for any of the machines supported by Whitesmiths, Ltd. compilers. This enhances uniformity of language implementation and simplifies programming for more than one target machine.

BUGS

On systems where the code generator (p2) talks to an existing, non-Idris, assembler, many of these utilities are not provided; hence, much of this section may be irrelevant.

NAME

Conventions - using the utilities

FUNCTION

Each of the utilities described in this section is a separate "program" that may be run, or "invoked", by typing a "command line", usually in response to a "prompt" such as the Idris "% ". This document provides a systematic guide to the conventions that govern how command lines are specified. It also summarizes the layout of the individual utility descriptions that follow, so that you know what information appears where, and in what format.

Command Lines

In general, a command line has three major parts: a program name, an optional series of flags, and a series of non-flag arguments, usually given in that order. Each element of a command line is usually a string separated by whitespace from all the others. Most often, of course, the program name is just the (file) name of the utility you want to run. Flags, if given, change some aspect of what a utility does, and generally precede all other arguments, because a utility must interpret them before it can safely interpret the remainder of the command line.

The meaning of the non-flag arguments strongly depends on the utility being run, but there are five general classes of command lines, presented here (where possible) with examples taken from the portable software tools, since they run much the same way on many different systems:

1) program name and flags followed by a series of filenames. These programs (called filters) process each file named in the order specified. An example is sort.

2) program name and flags followed by a series of arguments that are not filenames, but strings to which the program gives some other interpretation. echo is one such utility.

3) program name and a mandatory argument, which precedes the rest of the command line, followed by flags and other arguments. grep and tr belong to this class.

4) program name and flags followed by a series of "source" filenames, and a single "destination" filename. A filename to be paired with each source name is created by applying "pathname completion" to the destination name; for instance, the destination filename might be a directory of files whose names match the source filenames. These programs (called directed utilities) then perform some operation using each pair of files. diff is one example.

5) program name and flags followed by a command line that the utility executes after changing some condition under which it will run. These tend to be more sophisticated tools, like error, which redirects error messages under Idris.

A summary of the command line classes looks like this:

Class	Example	Syntax
filter	sort	<progname> <flags> <files>
string arguments	echo	<progname> <flags> <args>
mandatory argument	grep	<progname> <arg> <flags> <files>
directed	diff	<progname> <flags> <files> <dest>
prefix	error	<progname> <flags> <command>

Note that, in general, <flags> are optional in any command line.

Flags

Flags are used to select options or specify parameters when running a program. They are command line arguments recognized by their first character, which is always a '-' or a '+'. The name of the flag (usually a single letter) immediately follows the leading '-' or '+'. Some flags are simply YES/NO indicators -- either they're named, or they're not -- but others must be followed by some additional information, or "value". This value, if required, may be an integer, a string of characters, or just one character. String or integer values are specified as the remainder of the argument that names the flag; they may also be preceded by whitespace, and hence be given as the next argument.

The flags given to a utility may appear in any order, and two or more may be combined in the same argument, so long as the second flag can't be mistaken for a value that goes with the first one. Some flags have only a value, and no name. These are given as a '-' or '+' immediately followed by the value.

Thus all of the following command lines are equivalent, and would pass to uniq the flags -c and -f:

```
% uniq -c -f
% uniq -f -c
% uniq -fc
```

And each of the following command lines would pass the three flags -c3, -n, and -4 to pr:

```
% pr -c3 -4 -n file1
% pr -4 -nc 3 file1
% pr -n4 -c 3 file1
```

In short, if you specify flags so that you can understand them, a utility should have no trouble, either.

Usually, if you give the same flag more than once, only the last occurrence of the flag is remembered, and the repetition is accepted without comment. Sometimes, however, a flag is explicitly permitted to occur multiple times, and every occurrence is remembered. Such flags are said to

be "stacked," and are used to specify a set of program parameters, instead of just one.

Another special flag is the string "--", which is taken as a flag terminator. Once it is encountered, no further arguments are interpreted as flags. Thus a string that would normally be read as a flag, because it begins with a '-' or a '+', may be passed to a utility as an ordinary argument by preceding it with the argument "--". The string "-" also causes flag processing to terminate wherever it is encountered, but, unlike "--", is passed to the utility instead of being "used up", for reasons explained below.

If you give an unknown flag to a utility, it will usually display a hint to remind you of what the proper flags are. This message summarizes the format of the command line the utility expects, and is explained below in the synopsis section of the psuedo-manual page. Should you forget what flags a utility accepts, you can force it to output this "usage summary" by giving it the flag "-help", which is never a valid flag argument. (If a utility expects a mandatory argument, you'll have to say "-help -help" to get past the argument.)

Finally, be warned that some combinations of flags to a given utility may be invalid. If a utility doesn't like the set you've given, it will output a message to remind you of what the legal combinations are.

Files

Any utility that accepts a series of input filenames on its command line will also read its standard input, STDIN, when no input filenames are given, or when the special filename "-" is encountered. Hence sort can be made to read STDIN just by typing:

```
% sort
```

while the following would concatenate file1, then STDIN, then file2, and write them to STDOUT:

```
% cat file1 - file2
```

Naturally, whenever STDIN is read, it is read until end-of-file, and so should not be given twice to the same program.

Manual Pages

The remainder of this document deals with the format of the manual pages describing each of the utilities. Manual pages are terse, but complete and very tightly organized. Because of their general spareness, getting information out of them hinges on knowing where to find what you're after, and what form it's likely to take when you find it. Manual pages are divided into several standard sections, each of which covers one aspect of using the documented utility. So, for clarity, the rest of this document is presented as a psuedo-manual page, with the remarks on each section of a real page appearing under the normal heading for that section.

NAME

name - the name and a short description of the utility

SYNOPSIS

This section gives a one-line synopsis of the command line that the utility expects. The synopsis is taken from the message that the flag -help will cause most utilities to output, and indicates the main components of the command line: the utility name itself, the flags the utility accepts, and any other arguments that may (or must) appear.

Flags are listed by name inside the delimiters "-[" and "]". They generally appear in alphabetic order; flags consisting only of a value (see above) are listed after all the others. If a flag includes a value, then the kind of value it includes is also indicated by one of the following codes, given immediately after the flag name:

Code	Kind of Value
*	string of characters
#	integer (word-sized)
##	integer (long)
?	single character

A '#' designates an integer representable in the word size of the host computer, which may limit it to a maximum as small as 32,767 on some machines. A "##" always designates a long (four-byte) integer, which can represent numbers over two billion. An integer is interpreted as hexadecimal if it begins with "0x" or "0X", otherwise as octal if it begins with '0', otherwise as decimal. A leading '+' or '-' sign is always permitted.

If a flag may meaningfully be given more than once (and stacks its values), then the value code is followed by a '^'.

Thus the synopsis of pr:

```
pr -[c# e# h l# m n s? t* w# +## ##] <files>
```

indicates that pr accepts eleven distinct flags, of which -c, -e, -l, and -w include word-sized integer values, -h, -m, and -n include no values at all, -t includes a string of characters, -s includes a single character, and the two flags +## and -## are nameless, consisting of a long integer alone.

Note that flags introduced by a '-' are shown without the '-'. Roughly the same notation is used in the other sections of a manual page to refer to flags. In the pr manual page, for example, -c# would refer to the flag listed above as c#, while -[c# e# w#] would refer to the flags "c# e# w#".

The position and meaning of non-flag arguments are indicated by "metanotions", that is, words enclosed by '<' and '>' (like <files> in the example above). Each metanotion represents zero or more argu-

ments actually to be given on the command line. When entering a command line, you type whatever the metanotion represents, and type it at that point in the line. In the example, you would enter zero or more filenames at the point indicated by <files>.

No attempt is made in this section to explain the semantics of the command line -- for example, combinations of arguments that are illegal. The next section serves that purpose.

FUNCTION

This section generally contains three parts: an overview of the operation of the utility, a description of each of its flags, then (if necessary) additional information on how various flags or other arguments interact, or on details of the utility's operation that affect how it is used.

Usually, the opening overview is brief, summarizing just what the utility does and how it uses its non-flag arguments. The flag descriptions following consist of a separate sentence or more of information on each flag, introduced by the same flag name and value code given under SYNOPSIS. Each description states the effect the flag has if given, and the use of its value, if it includes one. The parameters specified by flag values generally have default settings that are used when the flag is not given; such default values appear at the end of the description. Flags are listed in the same order as in the synopsis line.

Finally, one or more paragraphs may follow the flag descriptions, in order to explain what combinations of flags are not permitted, or how certain flags influence others. Any further information on the utility of interest to the general user is also given here.

RETURNS

When it finishes execution, every utility returns one of two values, called success or failure. This section states under what conditions a utility will return one rather than the other. A successful return generally indicates that a utility was able to perform all necessary file processing, as well as all other utility-specific operations. In any case, the returned value is guaranteed to be predictable; this section gives the specifics for each utility.

Note that the returned value is often of no interest -- it can't even be tested on some systems. But when it can be tested, it is instrumental in controlling command scripts.

EXAMPLE

Here are given one or more examples of how the utility can be used. The examples seek to show the use of the utility in realistic applications, if possible in conjunction with related programs.

Generally, each example is surrounded by explanatory text, and is set off from the text by an empty line and indentation. In each example, lines preceded by a prompt (such as "% ") represent user input; other lines are the utility's response to the input shown.

FILES

This section lists any external files that the utility requires for correct operation. Most often, these files are system-wide tables of information or common device names.

SEE ALSO

Here are listed the names of related utilities or tutorials, which should be examined if the current utility doesn't do quite what you want, or if its operation remains unclear to you. Another utility may come closer, and seeing the same issues addressed in different terms may aid your understanding of what's going on.

Other documents in the same manual section as the current page are simply referred to by title; documents in a different section of the same manual are referred to by title and section number.

BUGS

This section documents inconsistencies or shortcomings in the behavior of the utility. Most often, these consist of deviations from the conventions described in this manual page, which will always be mentioned here. Known dangers inherent in the improper use of a utility will also be pointed out here.

BUGS

There is a fine line between being terse and being cryptic.

NAME

ROM - writing read-only code

FUNCTION

The C compiler never generates self modifying code. If all instructions are steered into the text section at code generation time (as is the default), then the composite text section produced by the program linker can safely be made read only. This means that a program to be run under an operating system can have its text section write protected every time it is dynamically loaded.

This also means that a free-standing program can have its text section "burned" into ROM (Read Only Memory) for any of the myriad reasons that people find to do so. A free-standing program usually has other issues to contend with as well, however.

Each program has a data section, in addition to its text section. This contains all static and extern declarations, with their initial values. C provides no way to distinguish parameters and tables, which never vary during program execution, from variables with important initial values, from variables that need not be initialized. The safest thing to do, therefore, is to assume the entire data section consists of variables with important initial values -- and, therefore, that an initial image of the data section must be copied from someplace in ROM to the expected addresses in RAM (Random Access Memory, which is writable), at program startup.

This is not a major burden. At program startup, a free-standing C program must have its stack setup -- for autos, arguments, and function call linkages. It is just a bit more work to copy the initial data image. The universal link utility is easily instructed to relocate the data section for its eventual RAM destination, while the hex utility can place it following the text section in ROM. The link utility also can be instructed to define symbols at the end of the text section and the end of the data section in its output file. Thus an initialization routine to copy data from ROM to RAM can simply step through memory from the "end of text" symbol to the "end of data" symbol.

There are ways to decrease the amount of data that must be initialized this way, sometimes to none at all. First there are literals -- character strings, switch tables, and constants -- that the compiler generates of its own accord. On a machine that supports separate instruction and data spaces, literals are steered by default into the data section. Specifying the flag -x6 to the code generator encourages it to put literals in the text section. It is rare that a ROM-based system also uses separate instruction and data spaces.

Similarly, extensive tables that are known to be read only can be collected into one C source file and compiled with the flag -x7 to the code generator, thus placing all data declarations for that file in the text section.

If the remaining variables can be contrived to require no initialization at startup, or to require only a simple initialization, such as all zeros, then the data section can be simply discarded at ROM burning time. This

approach requires, naturally, that all of the source code be available for scrutiny.

One case where this is not possible is when library functions, available only in binary, are incorporated into a ROM-based, or shareable, product. Libraries provided by Whitesmiths, Ltd. contain no gratuitous barriers to being shared. A function such as exp, for instance, contains only tables in its data section -- it may safely be passed through the linker to have its data merged into its text section.

Some functions, however, of necessity contain static variables with important initial values. Usually these functions interact with operating system services and hence are not found in ROM-based programs. Examples are onexit/exit, which remembers the head of a list of functions to be called on program exit, and sbreak, which may hold the current end of the dynamic data area. Still other functions with static variables may find use in free standing programs -- examples are alloc/free and _when/_raise.

The best practice, therefore, is to adopt the most general startup initialization described above. It is often too hard to determine which library functions are safe, and even harder to remember special rules for keeping new code safe.

NAME

as.68k - assembler for MC68000

SYNOPSIS

as.68k -[a o* s x] <files>

FUNCTION

as.68k translates MC68000 assembly language to Whitesmiths format relocatable object images. Since the output of the C code generator p2.68k is assembly code, as.68k is required to produce relocatable images suitable for binding with link.

The flags are:

-a when assembling absolute memory references, use absolute short (16-bit) address mode, rather than absolute long. This amounts to a promise that the eventual executable image, in its entirety, will be less than 64k bytes in length.

-o* write the output to the file *. Default is xeq. Under some circumstances, an input filename can take the place of this option, as explained below.

-s emit 16-bit, program counter relative references to memory, wherever permitted by the hardware, including for external and forward references. Elsewhere, either absolute short or absolute long references are used, depending on whether -a was given. This option may be safely specified only if every symbol is defined within 32K bytes of all references to it.

-x place all symbols in the object image. Default is to place there only those symbols that are undefined or that are to be made globally known.

If <files> are present, they are concatenated in order and used as the input file instead of the default STDIN.

If -o is absent, and one or more <files> are present, and the first filename ends with ".s", then as.68k behaves as if -o were specified using the first filename, only with the trailing 's' changed to 'o'. Thus,

```
as.68k file.s
```

is the same as

```
as.68k -o file.o file.s
```

A relocatable object image consists of a 28-byte header followed by a text segment, a data segment, the symbol table, and relocation information. The header consists of the short int 0x992d, an unsigned short int containing the number of symbol table bytes, and six unsigned long ints, giving: the number of bytes of object code defined by the text segment, the number of bytes defined by the data segment, the number of bytes defined by the bss segment, two zeros, and the data segment offset (always

equal to the text segment size). All ints in the object image are written most significant byte first, in pure descending order of byte significance.

Text and data segments each consist of an integral number of short ints. The text segment is relocated relative to location zero, the data segment is relocated relative to the end of the text segment, and the bss segment is relocated relative to the end of the data segment.

Relocation information consists of two byte streams, one for the text segment and one for the data segment, each terminated by a zero control byte. Control bytes in the range [1, 31] cause that many bytes in the corresponding segment to be skipped; bytes in the range [32, 63] skip 32 bytes, plus 256 times the control byte minus 32, plus the number of bytes specified by the relocation byte following.

All other control bytes control relocation of the next short or long int in the corresponding segment. If the 1-weighted bit is set in the control byte, then a change in load bias must be subtracted from the int. The 2-weighted bit is set if a long is being relocated instead of a short; the symbol code is the control byte right shifted two places minus 16.

A symbol code of 47 is replaced by a code obtained from the byte or bytes following in the relocation stream. If the next byte is less than 128, then the symbol code is its value plus 47. Otherwise the code is that byte minus 128, times 256, plus 175 plus the value of the next relocation byte after that one.

A symbol code of zero calls for no further relocation; 1 means that a change in text bias must be added to the item (word or longword); 2 means that a change in data bias must be added; 3 means that a change in bss bias must be added. Other symbol codes call for the value of the symbol table entry indexed by the symbol code minus 4 to be added to the item.

Each symbol table entry consists of a value longword, a flag byte, and an eleven-byte name padded with trailing NULs. Meaningful flag values are 0 for undefined, 4 for defined absolute, 5 for defined text relative, 6 for defined data relative, and 7 for defined bss relative. To this is added 010 if the symbol is to be globally known. The value of an undefined symbol is the minimum number of bytes that must be reserved, should there be no explicit defining reference.

SEE ALSO

link, p2.68k, rel

BUGS

Due to limitations in the MC68000 itself, neither form of 16-bit memory reference is as useful as one might wish. The hardware disallows storing into a location accessed by a PC-relative address mode, obliging the assembler not to use one where it would be unpalatable. Sixteen-bit absolute references are sign-extended, which can lead to surprises in accessing the upper half of the address space; as.68k, of course, can do nothing to alleviate them.

NAME

c - multi-pass command driver

SYNOPSIS

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

FUNCTION

c is designed to facilitate multi-pass operations such as compiling, assembling, and linking source files, by expanding commands from a prototype script for each of its file arguments. It is best described by example. A script for Pascal compilation on the PDP-11 might look like:

```
p:/etc/bin/ptc      ptc
c:/etc/bin/pp       pp -x -i/lib/
:/odd/p1            p1 -cem
:/odd/p2.11         p2
s:/etc/bin/as       as
o::/etc/bin/link    link -lp.11 -lc.11 /lib/Crts.o
```

c is intended for installation under multiple aliases. Each alias normally implies the name of a ".proto" file containing a corresponding script, to be searched for in the same way that the shell looks for the file named by a command. That is, if the alias contains a '/', then only the directory named is searched; otherwise, the search is of each directory in the current execution path. If the script above were named pc.proto, it could be implicitly invoked by running c under the alias pc. The -f flag may be used to override this lookup procedure.

By convention, the alias c is a synonym for the host machine native C compiler driver, and pc for the host Pascal driver.

Given the script above, c would produce a ".o" file for any argument files ending in ".p" by executing the following command list:

```
  COMMAND          ARGUMENTS
/etc/bin/ptc    ptc -o T1 file.p
/etc/bin/pp     pp -o T2 -x -i/lib/ T1
/odd/p1         p1 -o T1 -cem T2
/odd/p2.11      p2 -o T2 T1
/etc/bin/as     as -o file.o T2
```

where file.p is the argument file, and T1 and T2 are temporary intermediates.

Files whose names end in ".c" would be processed beginning at the script line labelled "c:"; similarly, ".s" files would be processed starting at the line labelled "s:". In general, all ".o" files produced, plus any files whose suffix matches no line prefix, are collected and passed to the link program for final binding. A file is passed to the link program only if its name is not already in the argument vector of the link line. Any error reported by any of the processing programs terminates processing of the current file; if any files are terminated the link program is not run.

The flags are:

-f* take prototype input from file * instead of from the ".proto" file implied by the command line alias. If * is "-", STDIN is read.

-o* place the output of the link line program in file * (by prepending -o* to its argument list). The prepending occurs only if this flag is given; naturally, it should be given only if the program will accept a -o specification.

-p* prefix the names of all permanent output files with the pathname *. If the -o flag is given, then the link line output filename will also be prefixed with *, if it doesn't already contain a '/'. All other output files will be stripped of any pre-existent pathname before being prefixed.

-v before executing each command, output its arguments to STDOUT. The default is to output only the name of each processed file and the name of the linker when (and if) it is run, each name followed by a colon and a newline.

+* stop production at prototype line whose prefix is *. The string given is matched against the prefix fields of each prototype line, and execution of prototype commands halts after the line preceding the matching one. In the example above, +c would cause ".p" files to be converted to ".c" files, with no subsequent processing, +s would process ".p" or ".c" files to ".s", and +o would inhibit linking. Any file that results in no processing causes an error message.

Each line in the prototype file has up to four parts: a prefix string (perhaps null) terminated with a colon, a pathname specifying a program to be run, and one or two groups of up to 16 strings, the groups separated by a colon. An argument vector is constructed for each program from the first string in the first group, a -o specification, the remaining strings in the first group, and one of the argument files. The second group of strings, if present, is appended to the vector after the argument file. Each entry on the command line is delimited by arbitrary whitespace. The final line in the file may have a prefix field (and colon) alone, in which case it specifies only the output file suffix for the preceding line. The final line may also have a prefix field terminated with a double colon, defining it as a "link line" with the special characteristics noted above.

Note that a prototype line with a null prefix field cannot be specified by the user as an entry or exit point.

If the last line of a prototype is not a prefix only and not a link line, then its output is written to a temporary file, which is discarded.

RETURNS

c returns success if it succeeded in passing all of its command line files to at least one of the prototype programs (the link line program included), and if all of the executions it supervised returned success.

EXAMPLE

A prototype file to do C syntax checking only, with no code generation, might be:

```
c:/etc/bin/pp  pp -x -i/lib/
:/odd/p1       p1
```

This prototype would produce no output files, since the output of p1 would go to a temporary file.

BUGS

Needs some way to specify flags to the prototype programs.

This utility is currently provided for use only on Idris/UNIX host systems.

NAME

cpm - maintain CP/M diskettes

SYNOPSIS

cpm -[+cpm dec b c d p r t v x f*] <files>

FUNCTION

cpm reads and writes CP/M compatible filesystems written on standard eight-inch, soft-sectored, IBM compatible diskettes. Such filesystems administer a diskette by dividing it into 243 clusters of 1024 bytes; up to 16 clusters may be allocated to an extent; up to 64 extents may be entered into the directory. A file consists of one or more extents, numbered consecutively from zero. All extents belonging to the same file have the same eight-character name and three-character type; names and types are best limited to letters of one case and digits, for maximum portability.

The flags are:

+cpm Convolve the sectors on each track by the recipe:

```
sec = (sec * 6 + (13 <= sec)) % 26;
```

where sec is in the interval [0,26), i.e. one less than the hardware sector number. This option is necessary on those machines that write diskette sectors without convolving, i.e. starting with track 0 sector 1 for seek address 0L and taking subsequent sectors sequentially.

-dec Deconvolve the sectors to correct for DEC convolution. The DEC recipe is:

```
sec = ((sec << 1) + (13 <= sec) + trk * 6) % 26;
trk =+ 1;
```

where sec is as above, and trk is in the interval [0,77). This option is necessary on DEC systems such as RT-11 and RSX-11M; the CP/M reconvolving also takes place as above.

-b Treat all files to be copied as binary images. Default is text mode, with possible processing of carriage returns, nulls, control-Z codes, etc.

-c Create new diskette with empty directory, then copy named files to diskette.

-d Delete named files from diskette.

-f* Diskette is accessed as filename *. Default is "dx0:" if -dec flag is present, else "/dev/rx0".

-p Print files, i.e. copy them from diskette to STDOUT. If no files are specified, all files are printed.

-r Replace named files on diskette.

-t Tabulate the diskette directory, showing all allocated extents in alphabetical order.

-v Be verbose about operations, naming files deleted "d:", added "a:", or extracted "x:". List number of free extents and clusters if -t is specified. Default is reasonably silent operation, save for errors.

-x Extract files, i.e. copy them from diskette to files with the same name. If no files are specified, all files are extracted.

At most one of the flags -c, -d, -p, -r, -t, -x may appear; default is -t.

RETURNS

cpm returns success if the directory structure of the diskette is intact and no file transfers failed, else failure.

EXAMPLE

To create a diskette and check it, under a DEC operating system:

```
>cpm -dec -c cscrip.cmd lscrip.cmd
>cpm -dec -rb prog.com
>cpm -dec
>cpm -dec -p cscrip.cmd
```

NAME

db - binary file editor/debugger

SYNOPSIS

db -[a c n p u] <file>

FUNCTION

db is an interactive editor designed to work with binary files. It operates in one of three modes: 1) as a binary file editor, for inspecting and patching arbitrary direct access files, such as disk images or encoded data files; 2) as a "prepartum" debugger, for inspecting and patching executable or relocatable files; and 3) as a "postmortem" debugger, for inspecting core images produced by the system, possibly using information from the original executable file. In the latter two cases, db recognizes the structure of object, executable, and core files, and does special interpretation of addresses to simulate the runtime placement of code and data.

db contains a disassembler enabled by specifying the mode 'm' in the o, p, or l commands (also used by the d and s commands, and in context searching). The disassembler interprets the file as machine instructions and translates them into conventional symbolic notation, using a symbol table and relocation information, when available, to output addresses. This "machine mode" is the initial default used for files not flagged as absolute.

A different disassembler is incorporated into db for each target machine to be supported. Thus, there may be variations with names like db11, db80, etc. By convention, the debugger for the host machine is called db.

The flags are:

-a treat <file> as an absolute binary file.

-c use the file core along with <file> and operate in debug mode. core will be the initial current file.

-n suppress the output of the current byte number normally preceding each item displayed.

-p write a prompt character '>' to STDOUT before reading command input. By default, no prompt is output.

-u open <file> in update mode. By default, <file> is opened for reading only.

At most one of -a and -c may be specified. Any core image file must always be named core. If <file> is not specified, xeq is assumed.

db has a great deal in common with the text editor e. It operates on the current binary file, a corresponding core image, or both, by reading from STDIN a series of single-character commands, and writing any output to STDOUT. Just as commands in e may be preceded by line addresses, db commands may be preceded by byte addresses, the syntax of which is a superset

of e address syntax. Analogous commands in the two editors have the same name; default behavior is also similar.

db imposes no structure on the data contained in its input files, except that implied by the current input/output mode, which defines the length of the "items" on which many commands operate. Outside machine mode, an item is one, two, or four bytes of the file. In machine mode, an item is the instruction disassembled starting at the current location, and is of varying length. Byte ordering of multiple-byte items is determined by the target machine, as are other machine-dependent characteristics such as symbol table structure and the size of an int. Thus, examining a file created for some machine other than the target is inadvisable. If <file> cannot be interpreted as an object file in standard format for the target machine, it is treated as absolute. If a core file does not begin with control bytes appropriate for the target, db aborts.

Address Interpretation and Syntax

In an absolute file, a byte address is simply the location of a byte relative to the start of the file (the first byte having address zero). In object or core files, an address consists of two components: the memory location a byte would occupy if the current file were actually loaded, and a flag indicating whether the location is within the text segment or within the data/bss segment of the program. (The program's bss segment immediately follows its data segment.) The ranges of addresses in the two segments may overlap; an address that could refer to either space is disambiguated as explained below.

A byte address is a series of one or more of the following terms:

> The character '.' addresses the "current byte" (or "dot"). Typically, dot is the first byte of the last item affected by the most recent command (but see the command descriptions below for specifics). Dot also refers to the same segment as that item.
>
> The character '$' addresses the last byte of the segment to which dot currently refers. For an absolute file, '$' has no meaning. A request to examine or change a given address will result in a seek to that byte of the file; the request succeeds if the subsequent I/O call does.
>
> In an object file, the characters 'T', 'D', and 'B' are constants that have the value of the first address in the text, data, and bss segments, respectively, and also have the appropriate segment component. In a core file, 'T' and 'D' have the same meaning, while 'B' is set to the address of the top of stack at the time of the dump, as recorded in the core file panel. All three have the value zero if the current file is absolute.
>
> An integer n addresses the nth byte of an absolute file, where bytes are numbered from zero. In an object or core file, an integer n addresses the nth byte of "memory", and the segment specified by the most recent term with an explicit segment component (or that of dot, if no such term exists). The easiest way to address the nth byte of

one of the two segments is to add an integer n to one of the constants 'T', 'D' or 'B'. An integer is interpreted as octal if it begins with '0', as hexadecimal if it starts with "0x" or "0X", and as decimal otherwise.

"'x", a single-quote followed by any lower-case letter, addresses the byte that has been previously assigned that letter, as described in the k command below.

A regular expression, enclosed in '/', causes a context search of the current segment (or of the whole file, if it is absolute), starting with the item after the current byte and proceeding toward the end of the current segment (or file). Syntax for regular expressions is the same as that for e, i.e., as implemented by the portable C library routines amatch, match, and pat. Each item in the region searched is converted to the current output format, and compared with the search string given. The search stops at the first matching item, the value of the term being the address of that item. An error is signaled if the search fails.

A regular expression enclosed in '%' or '?' causes a backward search. The context search starts with the item before the the current byte, and proceeds toward the start of the current segment (or of the file, if it is absolute), as described above. A backward search is impossible if the current output mode is machine mode.

A string beginning with a '_' or a '"' is taken to be a symbol name, terminated by whitespace or by one of the operators given below. A leading '_' is taken as part of the name; a leading '"' introduces names not starting with a '_', and is discarded. The value of the term is the address of the symbol; the segment referenced by the term is the one in which the symbol is defined.

A term may be followed by an arbitrary number of '*', each of which causes the value calculated to its left to be treated as a pointer into the data segment. The contents of the int-sized location pointed at then become the value of the term. Postfixing a term with '>' causes it to be analogously treated as a pointer into the text segment.

An address is resolved by scanning terms from left to right. Two terms separated by a plus '+' (or a minus '-'), resolve to a term that is the sum (or difference) of the values of the two original terms. Two successive terms not separated by an operator are summed. If a term explicitly refers to one of the two segments, then that segment becomes the segment to which the entire address refers.

If an address begins with a '+' or '-', then dot is assumed to be a term at the left of the '+' or '-'. The symbol '^' is equivalent to '-' in this context. Thus, "^3" standing alone is taken to be the same as ".-3".

A '+' ('-' or '^') not followed by a term is taken to mean +1 (-1). Thus '-' alone stands for ".-1", "++" stands for ".+2", "6---" stands for "3" and so on.

An address may be arbitrarily complex, so long as it yields a value within the bounds of the segment it refers to. (Again, for an absolute file, any address will be accepted.) There can be any number of addresses preceding a command so long as the last one or two are legal for that command. For a command requiring two addresses, it is an error for the second address to be less than the first, or for the addresses to refer to different segments. In debug mode, when pointing at the core file, stack addresses are accepted.

In machine instruction mode, addresses should be specified with care. db will disassemble starting at the address specified, but if the address is not the beginning of a valid instruction, the output is meaningless. To insure valid disassembly, start at the beginning of the text section or at the address of a text-relative symbol.

Command Syntax

Each db command consists of a single character, which may optionally be preceded by one or more addresses typically specifying the inclusive range of bytes in the file that the command is to affect. Addresses are separated by a comma ',' or by a colon ':'. If a colon is used, then dot is set to the location and segment specified by the address to the left of the colon before the rest of the command is interpreted. The command character may be followed by additional parameters. Multiple commands may be entered on a line by separating them with semicolons. Every command needing addresses has default values, so addresses can often be omitted; db complains if an address is referred to that doesn't exist.

In the command descriptions below, addresses in parentheses are the default byte addresses that will be used for the command if none are entered. The parentheses are not to be entered with the command, but are present as a syntactical notation. If a command is described without parentheses, no addresses are required. For a two-address command, supplying only one address will default the second to be identical with the one supplied. It is an error to precede with addresses commands that do not require any. The commands are:

(fp)a - display auto stack frame. Valid only if core is the current file. db keeps a pointer fp to the current stack frame, initialized to the one specified by the panel dumped to the core file. "a" may be preceded by an address which will be interpreted (and memorized) as the address of an auto stack frame.

b(1) - backup one or more stack frames. Valid only if core is the current file. "b" may be followed by a zero, which reinitializes the current stack frame pointer fp to the one in the panel, or by a count of the number of frames to be backed over. "b" and this count may also be followed by "a", which will backup one stack frame and display the auto stack frame values.

c - display the panel, which consists of the saved registers and cause of death. The registers are displayed in hexadecimal and symbolically, if meaningful. Valid only if core is the current file.

(.,.)d - print items that differ between executable and core files.

e file - enter a new modifiable data file, in same mode as before, then memorize filename. This works with -a flag only.

f - display name of modifiable data file.

(.)kx - mark address by memorizing it in x, where x is any lower case letter, suitable for later recall as a pointer with 'x.

(.,.)l<mode> - look at address specified, i.e., display item in output text representation, but don't change '.' to the address given. If <mode> is present, it changes the mode that was specified in the last 'o' command.

(.)n - display address as symbol plus offset.

o<mode> - change output text representation to <mode>, where <mode> is 'm', indicating disassembly of machine instructions, or any of the combinations [a|h|o|u] [c|s|i|l], for ASCII, hexadecimal, octal, or unsigned display of char, short, int, or long integers. If <mode> is not specified, the mode is set to the default used at db invocation: 'm' if the -a flag was not given, and unsigned short otherwise.

(.,.)p<mode> - print item in output text representation, then change '.' to point at last address specified. If <mode> is present, it changes the mode that was specified in the last 'o' command.

q - quit.

(.)r file - read binary contents of specified file and replace target file from current byte, extending the file as necessary. This is valid only with -a flag. file must not be the <file> being db'ed.

(.,.)s/pattern/new/[g] - for all items in range, expand to output text representation, edit text by changing instances of pattern to new, then update item using edited text pattern. If g suffix is specified, all matches in an item are altered, otherwise only the leftmost is altered.

(.)u - enter update mode, i.e., treat each subsequent line of input as the output text representation of an item and replace the addressed item. Successive items are replaced, extending the file as necessary, until an input line containing only a '.' is encountered. This does not work in 'm' mode. The input must be entered in the current mode, one item per line; otherwise db will complain.

(0,$)w file - write the specified range of the file being db'ed to file. file must not be the same as <file> being db'ed. This command is valid only with -a flag.

z file - point db at file. If file is omitted, z reports which file, core or executable, is currently pointed at.

(.)<newline> - equivalent to entering ". + size", where size is the number of bytes being displayed in the current mode. This form can be used to view one instruction or item at a time. If a byte address precedes the <newline>, the addressed item will be displayed. Dot will be set to the displayed byte.

!<command> - send all characters on the line to the right of the '!' to the system, to be interpreted as a system command. "!cd" is handled by db directly. Occurrences of the sequence "\f" will each be replaced by the currently remembered file before the transfer to system level is made. Dot is not changed by this command. Available only in Idris/UNIX systems.

= - display the byte number of dot.

On Idris/UNIX systems, if an ASCII DEL is typed, db interrupts its current task, if any, and displays '?'. It then will be ready to accept db commands again. Dot is generally ill-defined at this point.

SEE ALSO

link, rel

BUGS

This utility is currently provided for use only on Idris/UNIX host systems.

NAME

hex - translate object file to ASCII formats

SYNOPSIS

hex -[db## dr h m* r## s tb## +# #] <ifile>

FUNCTION

hex translates executable images produced by link to Intel standard hex format or to Motorola S-record object file format. The executable image is read from <ifile>. If no <ifile> is given, or if the filename given is "-", the file xeq is read.

The flags are:

-db## when processing a standard object file as input, override the object module data bias with ##.

-dr output text records as record type 0x81, data records as record type 0x82, for use with the Digital Research genemd utility under CP/M-86.

-h don't produce start record for Intel hex files.

-m* insert the string *, instead of <ifile>, into the Motorola S0 record generated when -s is given.

-r## interpret the input file as "raw" binary and not as an object file. Output is produced as described below, in either format, except that the starting address of the module is specified by the long integer ##.

-s produce S-records rather than the default hex format.

-tb## when processing a standard object file as input, override the object module text bias with ##.

+# start output with the #th byte. # should be between 0 and one less than the value specified by the -# flag. -4 +3 produces bytes 3, 7, 11, 15, ...; -2 +0 produces all even bytes; -2 +1 outputs all odd bytes; 0 is the default, which outputs all bytes.

-# output every #th byte. -2 outputs every other byte, -4 every fourth; 1 is the default, which outputs all bytes.

Output is written to STDOUT. When the input file is a standard object file, the load offset of the first record output for its text or data segment is determined as follows. If an offset is explicitly given for that segment (with "-db##" or "-tb##"), it is used. Otherwise, if all bytes of the input file are being output, the appropriate bias is used from the object file header. Otherwise, a load offset of zero is used. Then, the value given with "+#" is added to the chosen offset to obtain the first offset actually output.

Hex Files

A file in Intel hex format consists of the following records, in the order listed:

1) a '$' alone on a line to indicate the end of the (non-existent) symbol table. If -h is specified, this line is omitted.

2) data records for the text segment, if any. These represent 32 image bytes per line, possibly terminated by a shorter line.

3) data records for the data segment, if any, in the same format as the text segment records.

4) an end record specifying the start address.

Data records each begin with a ':' and consist of pairs of hexadecimal digits, each pair representing the numerical value of a byte. The last pair is a checksum such that the numerical sum of all the bytes represented on the line, modulo 256, is zero. The bytes on a data record line are:

a) the number of image bytes on the line.

b) two bytes for the load offset of the first image byte. The offset is written more significant byte first so that it reads correctly as a four-digit hexadecimal number.

c) a zero byte "00".

d) the image bytes in increasing order of address.

e) the checksum byte.

An end record also begins with a ':' and is written as digit pairs with a trailing checksum. Its format is:

a) a zero byte "00".

b) two bytes for the start address, in the same format as the load offset in a data record. The start address used is the first load offset output for the file.

c) a one byte "01".

d) the checksum byte.

S-Record Files

A file in Motorola S-record format is a series of records each containing the following fields:

<S field><count><addr><data bytes><checksum>

All information is represented as pairs of hexadecimal digits, each pair representing the numerical value of a byte.

<S field> determines the interpretation of the remainder of the line; valid S fields are "S0", "S1", "S2", "S8", "S9". <count> indicates the number of bytes represented in the rest of the line, so the total number of characters in the line is <count> * 2 + 4.

<addr> indicates the byte address of the first data byte in the data field. S0 records have two zero bytes as their address field; S1 and S2 records have <addr> fields of two and three bytes in length, respectively; S9 and S8 records have <addr> fields of two and three bytes in length, respectively, and contain no data bytes. <addr> is represented most significant byte first.

The S0 record contains the name of the input file, formatted as data bytes. If input was from xeq, XEQ is used as the name. S1 and S2 records represent text or data segment bytes to be loaded. They normally contain 32 image bytes, output in increasing order of address; the last record of each segment may be shorter. The text segment is output first, followed by the data segment. S9 records contain only a two-byte start address in their <addr> field; S8 records contain a three-byte address. The start address used is the first load offset output for the file.

<checksum> is a single byte value such that the numerical sum of all the bytes represented on the line (except the S field), taken modulo 256, is 255 (0xFF).

RETURNS

hex returns success if no error messages are printed, that is, if all records make sense and all reads and writes succeed; otherwise it reports failure.

EXAMPLE

The file hello.c, consisting of:

```
char *p {"hello world"};
```

when compiled produces the following Intel hex file:

```
% hex hello.o
$
:0E000000020068656C6C6F20776F726C640094
:00000001FF
```

and the following Motorola S-records:

```
% hex -s hello.o
S00A000068656C6C6F2E6F44
S111000002006865 6C6C6F20776F726C640090
S9030000FC
```

SEE ALSO

link, rel

NAME

lib - maintain libraries

SYNOPSIS

lib <lfile> -[c d i p r t v3 v6 v7 v x] <files>

FUNCTION

lib provides all the functions necessary to build and maintain object module libraries in either standard library format or those used by UNIX/System III, UNIX/V6, or UNIX/V7. lib can also be used to collect arbitrary files into one bucket. <lfile> is the name of an existing library file or, in the case of replace or create operations, the name of the library to be constructed.

The flags are:

-c create a library containing <files>, in the order specified. Any existing <lfile> is removed before the new one is created.

-d delete from the library the zero or more files in <files>.

-i take <files> from STDIN instead of the command line. Any <files> on the command line are ignored.

-p print named files (do a -x to STDOUT). If no files are named, all files are extracted.

-r in an existing library, replace the zero or more files in <files>; if no library <lfile> exists, create a library containing <files>, in the order specified. The files in <files> not present in the library are appended to it, in the order specified.

-t list the files in the library in the order they occur. If <files> are given, then only the files named and present are listed.

-v3 create the output library in UNIX/System III format, VAX-11 byte-order. This flag is meaningful with -c or -r; it is mandatory for -t and -x (UNIX/V7 format is assumed if this flag is not specified).

-v6 create the output library in UNIX/V6 format. Meaningful only with -c or -r.

-v7 create the output library in UNIX/V7 format, PDP-11 byte-order. Meaningful only with -c or -r.

-v be verbose. The name of each object file operated on is output, preceded by a code indicating its status: "c" for files retained unmodified, "d" for those deleted, "r" for those replaced, and "a" for those appended to <lfile>. If -v is used in conjunction with -t, each object file is listed followed by its length in bytes.

-x extract the files in <files> that are present in the library into discrete files with the same names. If no <files> are given, all files in the library are extracted.

At most one of the flags -[c d p r t x] may be given; if none is given, -t is assumed. Similarly, at most one of the flags -[v3 v6 v7] should be present; if none is present, standard library file format is assumed when a new library is created by -r or -c. An existing library is processed in its proper mode, except for UNIX/III libraries, for which the flag -v3 must be specified.

The standard library format consists of a two-byte header having the value 0177565, written less significant byte first, followed by zero or more entries. Each entry consists of a fourteen-byte name, NUL padded, followed by a two-byte unsigned file length, also stored less significant byte first, followed by the file contents proper. If a name begins with a NUL byte, it is taken as the end of the library file.

Note that this differs in several small ways from UNIX/V6 format, which has a header of 0177555, an eight-byte name, six bytes of miscellaneous UNIX-specific file attributes, and a two-byte file length. Moreover, a file whose length is odd is followed by a NUL padding byte in the UNIX format, while no padding is used in standard library format.

UNIX/III and UNIX/V7 formats are characterized by a header of 0177545, a fourteen-byte name, eight bytes of UNIX-specific file attributes, and a four-byte length. The length is in VAX-11 native byte order for UNIX/III or in PDP-11 native byte order for UNIX/V7. Odd length files are also padded to even.

RETURNS

lib returns success if no problems are encountered, else failure. After most failures, an error message is printed to STDERR and the library file is not modified. Output from the -t flag, and verbose remarks, are written to STDOUT.

EXAMPLE

To build a library and check its contents:

```
% lib clib -r one.o two.o three.o
% lib clib -tv
```

SEE ALSO

link, lord, rel

BUGS

If all modules are deleted from a library, a vestigial file remains.

Modifying UNIX/III, UNIX/V6, or UNIX/V7 format files causes all attributes of all file entries to be zeroed.

lib doesn't check for files too large to be properly represented in a library (> 65,534 bytes).

NAME

link - combine object files

SYNOPSIS

link -[a bb## b## c db## dr# d eb* ed* et* h i l*^ o* r sb* sd* st* tb## tf# t u*^ x#] <files>

FUNCTION

link combines relocatable object files in standard format for any target machine, selectively loading from libraries of such files made with lib, to create an executable image for operation under Idris, or for stand alone execution, or for input to other binary reformatters.

The flags are:

-a make all relocation items and all symbols absolute. Used to prerelocate code that will be linked in elsewhere, and is not to be re-relocated.

-bb## relocate the bss (block started by symbol) segment to start at ##. By default, the bss segment is relocated relative to the end of the data segment. Once -bb## has been specified, the resulting output file is unsuitable for input to another invocation of link.

-b## set the stack plus heap size in the output module header to ##. Idris takes this value, if non-zero, as the minimum number of bytes to reserve at execution time for stack and data area growth. If the value is odd, Idris will execute the program with a smaller heap plus stack size, so long as some irreducible minimum space can still be provided by using all of available memory.

-c suppress code output (.text and .data), and make all symbols absolute. Used to make a module that only defines symbol values, for specifying addresses in shared libraries, etc.

-db## set data bias to the long integer ##. Default is end of text section, rounded up to required storage boundary for the target machine.

-dr# round data bias up to ensure that there are at least # low-order binary zeros in its value. Ignored if -db## is specified.

-d do not define bss symbols, and do not complain about undefined symbols. Used for partial links, i.e., if the output module is to be input to link.

-eb* if the symbol * is referenced, make it equal to the first unused location past the bss area.

-ed* if the symbol * is referenced, make it equal to the first unused location past the initialized data area.

-et* if the symbol * is referenced, make it equal to the first unused location past the text area.

-h suppress header in output file. This should be specified only for stand alone modules such as bootstraps.

-i take <files> from STDIN instead of the command line. Any <files> on the command line are ignored.

-l* append library name to the end of the list of files to be linked, where the library name is formed by appending * to "/lib/lib". Thus "-lc.11" produces "/lib/libc.11". Up to ten such names may be specified as flags; they are appended in the order specified.

-o* write output module to file *. Default is xeq.

-r suppress relocation bits. This should not be specified if the output module is to be input to link or executed under a version of Idris that relocates commands on startup.

-sb* if the symbol * is referenced, make it equal to the size of the bss area.

-sd* if the symbol * is referenced, make it equal to the size of the data area.

-st* if the symbol * is referenced, make it equal to the size of the text area.

-tb## set text bias to the long integer ##. Default is location zero.

-tf# round text size up by appending NUL bytes to the text section, to ensure that there are at least # low-order binary zeroes in the resulting value. Default is zero, i.e. no padding.

-t suppress symbol table. This should not be specified if the output module is to be input to link.

-u* enter the symbol * into the symbol table as an undefined public reference, usually to force loading of selected modules from a library.

-x# specify placement in the output module of the input .text and .data sections. The 2-weighted bit of # routes .text input, the 1-weighted bit, .data. If either bit is set, the corresponding sections are put in the text segment output; otherwise, they go in the data segment. The default value used, predictably, is 2.

The bss section is always assumed to follow the data section in all input files. Unless -bb## is given, the bss section will be made to follow the data section in the output file, as well. It is perfectly permissible for text and data sections to overlap, as far as link is concerned; the target machine may or may not make sense of this situation (as with separate instruction and data spaces).

The specified <files> are linked in order; if a file is in library format, it is searched once from start to end. Only those library modules

are included which define public symbols for which there are currently outstanding unsatisfied references. Hence, libraries must be carefully ordered, or rescanned, to ensure that all references are resolved. By special dispensation, flags of the form "-l*" may be interspersed among <files>. These call for the corresponding libraries to be searched at the points specified in the list of files. No space may occur after the "-l", in this usage.

File Format

A relocatable object image consists of a header followed by a text segment, a data segment, the symbol table, and relocation information.

The header consists of an identification byte 0x99, a configuration byte, a short unsigned int containing the number of symbol table bytes, and six unsigned ints giving: the number of bytes of object code defined by the text segment, the number of bytes of object code defined by the data segment, the number of bytes needed by the bss segment, the number of bytes needed for stack plus heap, the text segment offset, and the data segment offset.

Byte order and size of all ints in the header are determined by the configuration byte.

The configuration byte contains all information needed to fully represent the header and remaining information in the file. Its value val defines the following fields: ((val & 07) << 1) + 1 is the number of characters in the symbol table name field, so that values [0, 8) provide for odd lengths in the range [1, 15]. If (val & 010) then ints are four bytes; otherwise they are two bytes. If (val & 020) then ints in the data segment are represented least significant byte first, otherwise, most significant byte first; byte order is assumed to be purely ascending or purely descending. If (val & 040) then even byte boundaries are enforced by the hardware. If (val & 0100) then the text segment byte order is reversed from that of the data segment; otherwise text segment byte order is the same. If (val & 0200) no relocation information is present in this file.

The text segment is relocated relative to the text segment offset given in the header (usually zero), while the data segment is relocated relative to the data segment offset (usually the end of the text segment). Unless -bb## is given, the bss segment is relocated relative to the end of the data segment.

Relocation information consists of two successive byte streams, one for the text segment and one for the data segment, each terminated by a zero control byte. Control bytes in the range [1, 31] cause that many bytes in the corresponding segment to be skipped; bytes in the range [32, 63] skip 32 bytes, plus 256 times the control byte minus 32, plus the number of bytes specified by the relocation byte following.

All other control bytes control relocation of the next short or long int in the corresponding segment. If the 1-weighted bit is set in such a control byte, then a change in load bias must be subtracted from the int.

The 2-weighted bit is set if a long int is being relocated instead of a short int. The value of the control byte right-shifted two places, minus 16, constitutes a "symbol code".

A symbol code of 47 is replaced by a code obtained from the byte or bytes following in the relocation stream. If the next byte is less than 128, then the symbol code is its value plus 47. Otherwise, the code is that byte minus 128, times 256, plus 175 plus the value of the next relocation byte after that one.

A symbol code of zero calls for no further relocation; 1 means that a change in text bias must be added to the item (short or long int); 2 means that a change in data bias must be added; 3 means that a change in bss bias must be added. Other symbol codes call for the value of the symbol table entry indexed by the symbol code minus 4 to be added to the item.

Each symbol table entry consists of a value int, a flag byte, and a name padded with trailing NULs. Meaningful flag values are 0 for undefined, 4 for defined absolute, 5 for defined text relative, 6 for defined data relative, and 7 for defined bss relative. To this is added 010 if the symbol is to be globally known. If a symbol still undefined after linking has a non-zero value, link assigns the symbol a unique area, in the bss segment, whose length is specified by the value, and considers the symbol defined. This occurs only if -d has not been given.

RETURNS

link returns success if no error messages are printed to STDOUT, that is, if no undefined symbols remain and if all reads and writes succeed; otherwise it returns failure.

EXAMPLE

To load the C program echo.o under Idris/S11:

```
% link -lc.11 -t /lib/Crts.o echo.o
```

with separate I/D spaces, for UNIX:

```
% link -lc.11 -rt -db0 /lib/Crts.o echo.o; taout
```

or with read only text section for UNIX:

```
% link -lc.11 -rt -dr13 /lib/Crts.o echo.o; taout
```

And to load the 8080 version of echo under CP/M:

```
A:link -hrt -tb0x0100 a:chdr.o echo.o a:clib.a a:mlib.a
```

SEE ALSO

hex, lib, lord, rel

NAME

lord - order libraries

SYNOPSIS

lord -[c* d*^ i r*^ s]

FUNCTION

lord reads in a list of module names, with associated interdependencies, from STDIN, and outputs to STDOUT a topologically sorted list of module names such that, if at all possible, no module depends on an earlier module in the list. Each module is introduced by a line containing its name followed by a colon. Subsequent lines are interpreted as either:

defs - things defined by the module,

refs - things referred to by the module, or

other stuff - other stuff.

Refs and defs have the syntax given by one or more formats entered as flags on the command line. Each character of the format must match the corresponding character at the beginning of an input line; a ? will match any character except newline. If all the characters of the format match, then the rest of the input line is taken as a ref or def name. Thus, the format flag "-d0x????D " would identify as a valid def any line beginning with "0x", four arbitrary characters and a "D ", so that the input line "0x3ff0D _inbuf" would be taken as a def named "_inbuf".

The flags are:

-c* prepend the string * to the output stream. Implies -s. Each module name is output preceded by a space; the output stream is terminated with a newline. Hence, lord can be used to build a command line.

-d* use the string * as a format for defs.

-i ignore other stuff. Default is to complain about any line not recognizable as a def or ref.

-r* use the string * as a format for refs.

-s suppress output of defs and refs; output only module names in order.

Up to ten formats may be input for defs, and up to ten for refs.

If no -d flags are given, lord uses the default def formats: "0x?????????B ", "0x?????????D ", "0x?????????T ", "0x????B ", "0x????D ", "0x????T ". If no -r flags are given, lord uses the default ref formats: "0x?????????U " and "0x????U ". These are compatible with the default output of rel.

If there are circular dependencies among the modules, lord writes the name of the file that begins the unsorted list to STDERR, followed by the message "not completely sorted". In general, rearrangements are made only

when necessary, so an ordered set of modules should pass through lord unchanged.

RETURNS

lord returns success if no error messages are printed, otherwise failure.

EXAMPLE

To create an ordered library of object modules under Idris:

```
% rel *.o | lord -c"lib libx.a -c" | sh
```

To order a set of objects using UNIX nm:

```
% nm *.o > nmlist
% lord < nmlist -c"ar r libx.a" | \
-d"???????T " -d"??????D " -d"??????B " -r"??????U " | sh
```

SEE ALSO

lib, rel

NAME

p1 - parse C programs

SYNOPSIS

p1 -[a b# c e l m n# o* r# u] <file>

FUNCTION

p1 is the parsing pass of the C compiler. It accepts a sequential file of lexemes from the preprocessor pp and writes a sequential file of flow graphs and parse trees, suitable for input to a machine-dependent code generator p2. The operation of p1 is largely independent of any target machine. The flag options are:

-a compile code for machines with separate address and data registers. This flag implies -r6 (because currently used only in conjunction with the MC68000 code generator).

-b# enforce storage boundaries according to #, which is reduced modulo 4. A bound of 0 leaves no holes in structures or auto allocations; a bound of 1 (default) requires short, int and longer data to begin on an even bound; a bound of 2 is the same as 1, except that 4-8 byte data are forced to a multiple of four byte boundary; a bound of 3 is the same as 2, except that 8 byte data (doubles) are forced to a multiple of eight byte boundary.

-c ignore case distinctions in testing external identifiers for equality, and map all names to lowercase on output. By default, case distinctions matter.

-e don't force loading of extern references that are declared but never defined or used in an expression. Default is to load all externs declared.

-l take integers and pointers to be 4 bytes long. Default is 2 bytes.

-m treat each struct/union as a separate name space, and require x.m to have x a structure with m one of its members.

-n# ignore characters after the first # in testing external identifiers for equality. Default is 7; maximum is 8, except that values up to 32 are permitted for the VAX-11 code generator only.

-o* write the output to the file * and write error messages to STDOUT. Default is STDOUT for output and STDERR for error messages.

-r# assign no more than # variables to registers at any one time, where # is reduced modulo 7. Default is 3 register variables; values above 3 are currently acceptable only for the MC68000 code generator (3 data registers + 3 address registers maximum), and the VAX-11 code generator (6 registers maximum).

-u take "string" as array of unsigned char, not array of char.

If <file> is present, it is used as the input file instead of the default STDIN. On many systems (other than Idris/UNIX), the -o option and <file> are mandatory because STDIN and STDOUT are interpreted as text files, and hence become corrupted.

EXAMPLE

p1 is usually sandwiched between pp and some version of p2, as in:

```
pp -x -o temp1 file.c
p1 -o temp2 temp1
p2.11 -o file.s temp2
```

SEE ALSO

pp

BUGS

p1 can be rather cavalier about semicolons.

NAME

p2.68k - generate code for MC68000 C programs

SYNOPSIS

p2.68k -[ck dx n* o* p x#] <file>

FUNCTION

p2.68k is the code generating pass of the C compiler. It accepts a sequential file of flow graphs and parse trees from p1 and writes a sequential file of assembly language statements, suitable for input to an MC68000 assembler. Two versions of p2.68k are available for the MC68000; one that generates code compatible with the Motorola VERSAdos assembler, and one that is compatible with the Idris assembler as.68k.

As much as possible, the compiler generates free-standing code; but for those operations which cannot be done compactly, it generates inline calls to a set of machine-dependent runtime library routines. The MC68000 runtime library is documented in Section IV of this manual.

The flags are:

-ck enable stack overflow checking.

-dx disable double indexing.

-n* use the name * as the module name in the assembler output. Default is to use the first defined external encountered.

-o* write the output to the file * and write error messages to STDOUT. Default is STDOUT for output and STDERR for error messages.

-p emit profiler calls on entry to each function.

-x# map the three virtual sections, for Functions (04), Literals (02), and Variables (01), to the two physical sections Code (bit is a one) and Data (bit is a zero). Thus, "-x4" is for separate I/D space, "-x6" is for ROM/RAM code, and "-x7" is for compiling tables into ROM. Default is 6.

If <file> is present, it is used as the input file instead of the default STDIN. On many systems (other than Idris/UNIX), <file> is mandatory, because STDIN is interpreted as a text file, and hence becomes corrupted.

Files output from p1 for use with the MC68000 code generator should be generated with: -a signifying separate Address and Data registers; -l since pointers are long; and no -b# flag, since only the default value of "-b1" is acceptable. For use with the VERSAdos assembler, p1 should be run with the flags "-cn7" to check externals properly; "-n8" is preferred for the Idris assembler.

Wherever possible, labels in the emitted code are each followed by a comment which gives the source line from which the code immediately following obtains, along with a running count of the number of words of code produced for a given function body.

EXAMPLE

p2.68k usually follows pp and p1, as follows:

```
pp -x -o temp1 file.c
p1 -aln8 -o temp2 temp1
p2.68k -o file.s temp2
```

SEE ALSO

as.68k, p1

BUGS

Stack overflow checking is only approximate, since a calculation of the exact stack high water mark is not attempted.

Complex expressions involving the ternary operator (?:) may generate branch range errors. The only solution is to simplify the expression.

NAME

pp - preprocess defines and includes

SYNOPSIS

pp -[c d*^ i* o* p? s? x 6] <files>

FUNCTION

pp is the preprocessor used by the C compiler, to perform #define, #include, and other functions signalled by a #, before actual compilation begins. It can be used to advantage, however, with most language processors. The flag options are:

-c don't strip out /* comments */ nor continue lines that end with \.

-d* where * has the form name=def, define name with the definition string def before reading the input; if =def is omitted, the definition is taken as "1". The name and def must be in the same argument, i.e., no blanks are permitted unless the argument is quoted. Up to ten definitions may be entered in this fashion.

-i* change the prefix used with #include <filename> from the default "" to the string *. Multiple prefixes to be tried in order may be specified, separated by the character '|'.

-o* write the output to the file * and write error messages to STDOUT. Default is STDOUT for output and STDERR for error messages. On many systems (other than Idris/UNIX), the -o option is mandatory with -x because STDOUT is interpreted as a text file, and hence becomes corrupted.

-p? change the preprocessor control character from '#' to the character ?.

-s? change the secondary preprocessor control character from '@' to the character ?.

-x put out lexemes for input to the C compiler p1, not lines of text.

-6 put out extra newlines and/or SOH ('\1') codes to keep source line numbers correct for UNIX/V6 compiler or ptc.

pp processes the named files, or STDIN if none are given, in the order specified, writing the resultant text to STDOUT. Preprocessor actions are described in detail in Section I of the C Programmers' Manual.

The presence of a secondary preprocessor control character permits two levels of parameterization. For instance, the invocation

```
pp -c -p@
```

will expand define and ifdef conditionals, leaving all # commands and comments intact; invoking pp with no arguments would expand both @ and # commands. The flag -s# would effectively disable the secondary control character.

EXAMPLE

The standard style for writing C programs is:

```
/* name of program
 */
#include <std.h>

#define MAXN    100

COUNT things[MAXN];
etc.
```

The use of uppercase only identifiers is not required by pp, but is strongly recommended to distinguish parameters from normal program identifiers and keywords.

SEE ALSO

p1, ptc

BUGS

Unbalanced quotes ' or " may not occur in a line, even in the absence of the -x flag. Floating constants longer than 38 digits may compile incorrectly, on some host machines.

NAME

prof - produce execution profile

SYNOPSIS

prof -[a f +l n r s t +z] [<ofile>] [<pfiles>]

FUNCTION

prof correlates one or more files of statistics recorded during the execution of a program with the symbol table of the corresponding executable file, and produces a report profiling the run. The statistics are contained in "profile" files of standard format. The <ofile> given on the command line is the program whose execution produced the statistics.

Flags are:

-a sort output in increasing order of address.

-f sort output in decreasing order of frequency of calls to each function.

+l include local symbols in output. By default, only global symbols are included.

-n sort output in increasing order of symbol name.

-r reverse the default sense of the output sort.

-s suppress the default heading.

-t sort output in decreasing order of time spent in each function. This is the default sort key.

+z include symbols in output for which no entries or execution time were recorded. By default, these are excluded.

If no <ofile> is given, "xeq" is used as the object filename. If no <pfiles> are given, prof expects profiling data to come from the file "profil". However, if any filenames appear after <ofile> on the command line, they will be read for profiling data instead. If more than one file is given, prof will accumulate data from each in turn, and output a single report giving the totals from all data files read. Thus the results of several instrumented runs of a given program may be merged to reduce statistical error. Naturally, if <pfiles> are given <ofile> must be given as well.

Profiling involves two statistics: counting the calls made to each function in a program, and recording what portion of execution time is spent inside each function. In its report, prof outputs one line for each text-relative symbol in <file> (subject to +l and +z), showing the symbol name, the percentage of total execution time spent in the region between this symbol and the next higher one, time in this region in milliseconds, the number of calls made directly to this symbol's address, and the average amount of time recorded per call. Also, prof treats the final location available for function entry counts as an overflow location, in which

calls are counted to all functions that could not be given a location of their own. If this count is non-zero, prof outputs it, with the name "other syms".

Idris records execution time by periodically examining the PC of the executing program, using its value to index an array whose elements correspond to the permissible range of the PC. The selected array element is then incremented. Function entry counts are maintained by calls to a counting routine, which every Whitesmiths C compiler can be made to output by specifying the flag -p to the code generator. A function must be thus instrumented for an entry count to be maintained for it. The counting routine itself is specified at link time, as part of the runtime startup code.

RETURNS

prof returns success if all input files are readable and consistent with their expected format.

SEE ALSO

The profile file format description is in Section III of the Idris Programmers' Manual, while the runtime initialization code used to profile under Idris is documented along with the Idris system interface, in Section III.a of this 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.

BUGS

Sampling errors may occur unless the scaling factor used to record the PC location is no larger than the smallest boundary on which an instruction may begin; that is, a given array element may overlap function bodies, and a PC caught in one function may be treated as if it were caught in another. prof does what it can to compensate, by dividing the value of an overlapping array element between the functions involved.

This utility is currently provided for use only on Idris host systems.

NAME

ptc - Pascal to C translator

SYNOPSIS

ptc -[c f k m# n# o* r s#] <ifile>

FUNCTION

ptc is a program that accepts as input lines of Pascal text and produces as output a corresponding C program which is acceptable to the Whitesmiths, Ltd. C compiler. If <ifile> is present, it is taken as the Pascal program to translate; otherwise input is taken from STDIN.

The flags are:

-c pass comments through to the C program.

-f set the precision for reals to single precision (float). Default is double.

-k permit pointer types to be defined using type identifiers from outer blocks. Default is ISO standard, i.e., the type pointed to must be defined in the same type declaration as the pointer type definition.

-m# make # the number of bits in MAXINT excluding the sign bit, e.g., MAXINT becomes 32767 for -m15, 1 for -m1, etc. Default for MAXINT is 32766 [sic]. Acceptable values for # are in the range [0 , 32). Declaring MAXINT greater than the size of a pointer (16 or 32) will give unpredictable results. On a target machine with 16-bit pointers, # should be less than 16.

-n# Make # the number of significant characters in external names. Default is 8 character external names.

-o* write the C program to the file * and diagnostics to STDOUT. Default is STDOUT for the C program and STDERR for diagnostics.

-r turn off runtime array bounds checks.

-s# make # the number of bits in the maximum allowable set size, i.e., the size of all sets whose basetype is integer becomes the specified power of two. Acceptable values are in the range [0 , 32). Default is 8 (256 elements).

The CP/M operating system implementation, on the Intel 8080 and Zilog Z/80, restricts the acceptable value for # to the range [0,16]; for maximum portability, this restriction should be honored.

Identifiers are mapped to uppercase to keep from conflicting with those declared as reserved words in C. Moreover, structure declarations may be produced that contain conflicting field declarations; and declarations are present for library functions that may not be needed. All of these peccadillos are forgiven by the use of appropriate C compiler options.

RETURNS

ptc returns success if it produces no diagnostics.

BUGS

Complex set expressions can produce very long lines that are truncated by pp.

NAME

rel - examine object files

SYNOPSIS

rel -[d g i o s t u v] <files>

FUNCTION

rel permits inspection of relocatable object files, in standard format, for any target machine. Such files may have been output by an assembler, combined by link, or archived by lib. rel can be used either to check their size and configuration, or to output information from their symbol tables.

The flags are:

-d output all defined symbols in each file, one per line. Each line contains the value of the symbol, a code indicating to what the value is relative, and the symbol name. Values are output as the number of digits needed to represent an integer on the target machine. Relocation codes are: 'T' for text relative, 'D' for data relative, 'B' for bss relative, 'A' for absolute, or '?' for anything rel doesn't recognize. Lowercase letters are used for local symbols, uppercase for globals.

-g print global symbols only.

-i print all global symbols, with the interval in bytes between successive symbols shown in each value field. Implies the flags -[d u v].

-o output symbol values in octal. Default is hexadecimal.

-s display the sizes, in decimal, of the text segment, the data segment, the bss segment, and the space reserved for the runtime stack plus heap, followed by the sum of all the sizes.

-t list type information for this file. For each object file the data output is: the size of an integer on the target machine, the target machine byte order, whether even byte boundaries are enforced by the hardware, whether the text segment byte order is reversed from that of the data segment, and the maximum number of characters permitted in an external name. If the file is a library, then the type of library is output and the information above is output for each module in the library.

-u list all undefined symbols in each file. If -d is also specified, each undefined symbol is listed with the code 'U'. The value of each symbol, if non-zero, is the space to be reserved for it at load time if it is not explicitly defined.

-v sort by value; implies the -d flag above. Symbols of equal value are sorted alphabetically.

If no flags are given, the default is -[d u]; that is, all symbols are listed, sorted in alphabetical order on symbol name. If more than one of

the flags -[d s t u] is selected, then type information is output first, followed by segment sizes, followed by the symbol list specified with -d or -u.

<files> specifies zero or more files, which must be in relocatable format, or standard library format, or UNIX/V6 library format, or UNIX/V7 library format. If more than one file, or a library, is specified, then the name of each separate file or module precedes any information output for it, each name followed by a colon and a newline; if -s is given, a line of totals is also output. If no <files> are specified, or if "-" is encountered on the command line, xeq is used.

RETURNS

rel returns success if no diagnostics are produced, that is, if all reads are successful and all file formats are valid.

EXAMPLE

To obtain a list of all symbols in a module:

```
% rel alloc.o
0x00000074T _alloc
0x00000000U _exit
0x000001feT _free
0x000000beT _nalloc
0x00000000U _sbreak
0x00000000U _write
```

SEE ALSO

lib, link, lord

III.a. Idris System Interface Library

NAME

Interface - to Idris system

FUNCTION

Programs written in C for operation on the MC68000 under Idris are translated according to the following specifications:

external identifiers - may be written in both upper and lowercase. The first eight letters must be distinct. An underscore '_' is prepended to each identifier.

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

literal data - such as strings, double constants, and switch tables, are normally generated into .text.

initialized data - is normally generated into .data. External data names are published via .globl declarations.

uninitialized declarations - result in an .globl reference, one instance per program file.

function calls - are performed by

1) moving arguments onto the stack, right to left. Character and short data are widened to long integer, float is converted to double.

2) calling via "jsr _func".

3) popping the arguments off the stack.

Except for returned value, the registers d0, d1, d2, d6, d7, a0, a1, a2, and the condition codes are undefined on return from a function call. All other registers are preserved. The returned value is in d7 (char or short widened to long integer, long integer, and pointer to), or in d6-d7 (float widened to double, and double).

stack frames - are maintained by each C function, using a6 as a frame pointer. On entry to a function, the instruction 'link a6,#' will stack a6 leaving a6 pointing to the stacked a6, and will reserve # bytes for automatics. If more than 32,768 bytes of autos are specified, additional code is generated to reserve space on the stack. Any non-volatile registers used (d3, d4, d5, a3, a4, a5) are then stacked and, if the top of stack scratch cell is used, an additional four bytes are reserved (by either stacking d0 along with other registers or by modifying the stack pointer). Arguments are now at 8(a6), 12(a6), etc. and auto storage is on the stack at -1(a6) on down. To return, the nonvolatile registers are restored from the stack, a6 and a7 are retrieved with an "unlk a6" and control is returned via "rts". Note that the stack must be balanced on exit if the nonvolatile registers are to be properly restored.

data representation - short integers are stored as two bytes, more significant byte first. Long integers are stored as four bytes, in descending order of significance. Floating numbers are represented as for the proposed IEEE Floating Point Standard, four bytes for float, eight for double, and are stored in descending order of byte significance. The IEEE representations are: most significant bit is one for negative numbers, else zero; next eleven bits (eight for float) are the characteristic, biased such that the binary exponent of the number is the characteristic minus 1022 (126 for float); remaining bits are the fraction, starting with the 1/4 weighted bit. If the characteristic is zero, the entire number is taken as zero and should be all zero to avoid confusing some routines that take shortcuts. Otherwise there is an assumed 1/2 added to all fractions to put them in the interval [0.5, 1.0). The value of the number is the fraction, times -1 if the sign bit is set, times 2 raised to the exponent.

storage bounds - even byte storage boundaries must be enforced for multibyte data. 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.

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

c68k - compile and link C programs

SYNOPSIS

c68k -[f* o* p* v +*] <files>

FUNCTION

c68k is an instantiation of the generic C driver described in Section II, configured to compile C (with filenames *.c) or as.68k (*.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.68k(II), c(II), pc68k, p2.68k(II)

NAME

pc68k - compile and link Pascal programs

SYNOPSIS

pc68k -[f* o* p* v +*] <files>

FUNCTION

pc68k 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.68k (*.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), c68k, 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 image:

```
% 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 68000 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

III.b. VERSAdos System Interface Library

NAME

Interface - to VERSAdos system

FUNCTION

Programs written in C for operation on the MC68000 under VERSAdos are translated according to the following specifications:

external identifiers - may be written in both upper and lowercase, but only one case is significant. The first seven letters must be distinct. A '.' is prepended to each identifier.

function text - is normally generated into section 0 and is not to be altered or read as data. External function names are published via xdef declarations.

literal data - such as strings, double constants, and switch tables, are normally generated into section 0.

initialized data - is normally generated into section 1. External data names are published via xdef declarations.

uninitialized declarations - result in an xref reference, one instance per program file.

function calls - are performed by

1) moving arguments onto the stack, right to left. Character and short data are widened to long integer, float is converted to double.

2) calling via "jsr .func".

3) popping the arguments off the stack.

Except for returned value, the registers d0, d1, d2, d6, d7, a0, a1, a2, and the condition codes are undefined on return from a function call. All other registers are preserved. The returned value is in d7 (char or short widened to long integer, long integer, and pointer to), or in d6-d7 (float widened to double, and double).

stack frames - are maintained by each C function, using a6 as a frame pointer. On entry to a function, the instruction "link a6,#" will stack a6 leaving a6 pointing to the stacked a6, and will reserve # bytes for automatics. If more than 32,768 bytes of autos are specified, additional code is generated to reserve space on the stack. Any non-volatile registers used (d3, d4, d5, a3, a4, a5) are then stacked and, if the top of stack scratch cell is used, an additional four bytes are reserved (by either stacking d0 along with other registers or by modifying the stack pointer). Arguments are now at 8(a6), 12(a6), etc. and auto storage is on the stack at -1(a6) on down. To return, the nonvolatile registers are restored from the stack, a6 and a7 are retrieved with an "unlk a6" and control is returned via "rts". Note that the stack must be balanced on exit if the nonvolatile registers are to be properly restored.

data representation - short integers are stored as two bytes, more significant byte first. Long integers are stored as four bytes, in descending order of significance. Floating numbers are represented as for the proposed IEEE Floating Point Standard, four bytes for float, eight for double, and are stored in descending order of byte significance. The IEEE representations are: most significant bit is one for negative numbers, else zero; next eleven bits (eight for float) are the characteristic, biased such that the binary exponent of the number is the characteristic minus 1022 (126 for float); remaining bits are the fraction, starting with the 1/4 weighted bit. If the characteristic is zero, the entire number is taken as zero and should be all zero to avoid confusing some routines that take shortcuts. Otherwise there is an assumed 1/2 added to all fractions to put them in the interval [0.5, 1.0). The value of the number is the fraction, times -1 if the sign bit is set, times 2 raised to the exponent.

storage bounds - even byte storage boundaries must be enforced for multibyte data. 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, unless an explicit module name is given to the code generator. The module name is published via an idnt declaration.

NAME

Conventions - VERSAdos system subroutines

SYNOPSIS

```
#include <vdos.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 system header file, <vdos.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 data structures for use with VERSAdos.

Herewith the principal definitions:

CTRLZ - 032 (ctl-Z), end-of-file from a terminal.
FHS - the VERSAdos file services block and oft used values.
IOS - the VERSAdos I/O service block along with oft used bit values.
MAXCMD - the size of the longest command line to be passed to C.
RCB - struct rcb, the Whitesmiths, Ltd. file control block.
VEOF - 0xC2 VERSAdos end-of-file from a file.

NAME

c - compiling C programs

SYNOPSIS

=CHAIN C sfile

FUNCTION

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

The C source file is at sfile.c, where sfile is the name typed in the submit line. The relocatable image from the assembler is put at sfile.ro.

EXAMPLE

To compile test.c:

```
=CHAIN C TEST
```

SEE ALSO

clink, pc

FILES

sfile.t?

BUGS

There should be some way to pass the -m flag to p1, or some of the myriad flags acceptable to pp, other than by modifying the command file proper.

NAME

pc - compiling Pascal programs

SYNOPSIS

=CHAIN PC sfile

FUNCTION

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

The Pascal source file is at sfile.p, where sfile is the name typed in the submit line. The relocatable image from the assembler is put at sfile.ro.

EXAMPLE

To compile test.p:

```
=CHAIN PC TEST
```

SEE ALSO

c, clink

FILES

sfile.t?

BUGS

There should be some way to pass some of the myriad flags acceptable to ptc, other than by modifying the command file proper.

NAME

clink - link C programs

SYNOPSIS

=LINK CHDR/<files>,NAME,NAME;L=CLIB

FUNCTION

Programs written in C must be linked with certain object modules that implement the standard C runtime environment. The normal VERSAdos LINK program is used, as shown in the example above, which produces an executable file "name.lo", and map file "name.ll" from one or more object files in the list <files> that presumably were produced by the C compiler. Filenames in this list are separated by slashes '/'.

The top of memory space allocated by the linker (section 15) is used for the stack plus heap area, which is grown at task startup time by chdr. The size of this area in bytes is given by the value of _hsize which is an external reference. Its default value is 24K bytes. To reduce or increase this value, define the variable _hsize in one of the <files> and insure that it is statically initialized to the desired value.

chdr.ro is the module that gets control when name.lo is run. It calls the function _main(), described in the VERSAdos library, which redirects the standard input and output as necessary, calls the user provided main(ac, av) with the command arguments, and passes the value returned by main on to exit(). All of this malarkey can be circumvented if <files> contains a defining instance of _main().

<files> can be one or more object modules produced by the C compiler, or produced from Motorola Structured Assembler source that satisfies the interface requirements of C, as described in the C Compiler Runtime Library Manual. 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.

clib is a library containing all of the portable C library functions, plus those required for the VERSAdos interface. It should be scanned last.

EXAMPLE

To compile and run the program echo.c:

```
=CHAIN C ECHO.C
=LINK CHDR/ECHO,ECHO,ECHO;L=CLIB
=ECHO hello world!
hello world!
```

SEE ALSO

_hsize, c

NAME

chdr - C startup code

SYNOPSIS

=LINK CHDR/<files>;L=CLIB

FUNCTION

chdr is the code that gets control whenever a C program is started. It first sets the total stack and heap size to the value in the long integer _hsize (by growing section 15 to _hsize bytes), sets the stack to the top of memory, translates the command line to lowercase, and calls _main, passing its return value on to exit. It also assures that neither text nor data begin at absolute location zero.

If the heap cannot be grown, the program aborts itself with a code of -1.

SEE ALSO

_hsize, _main

NAME

_hsize - size of the stack plus heap area

SYNOPSIS

```
BYTES _hsize {0x6000};
```

FUNCTION

_hsize specifies the size in bytes of the stack and heap area for the program. This area is allocated in section 15 at task startup time.

The default value provided in clib is 24K bytes. To reduce or increase this value, define the variable _hsize in the C program proper and insure that it is statically initialized to the desired value.

SEE ALSO

clink

NAME

_main - setup for main call

SYNOPSIS

```
BOOL _main()
```

FUNCTION

_main is the function called whenever a C program is started. It obtains the command line from parameters placed in _regs[] at task startup, 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 spaces. 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 the program name as passed to it by VERSAdos.

EXAMPLE

To avoid the loading of _main and all the 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.

SEE ALSO

exit

NAME

_pname - program name

SYNOPSIS

```
TEXT *_pname;
```

FUNCTION

_pname is the name by which the program is invoked. It is set from the command line at startup time and used for error printouts. Some portable programs provide a default program name by defining _pname, for those systems that are incapable of determining the actual invocation name; but not all programs are obliged to provide a defining instance of _pname. Hence this library module is provided, containing the default string "error".

NAME

to86 - convert to CP/M-86 or DOS executable format

SYNOPSIS

to86 -[b# cgx# dgx# exe o* s zb zh] <file>

FUNCTION

to86 translates standard 8086 object files to CP/M-86 ".cmd" or DOS ".exe" format files.

The flags are:

-b# override the stack plus heap size in the standard object file and set it to #. to86 takes this value as the minimum number of bytes to reserve beyond the end of the data group for stack and data area growth. If # is odd, the maximum space possible is reserved.

-cgx# for a CP/M-86 ".cmd" file, use # as the maximum group size ("G-Max") for the code group (or the combined code/data group). Default is zero [sic].

-dgx# for a CP/M-86 ".cmd" file, use # as the maximum group size ("G-Max") for the data group. Default is zero.

-exe produce a DOS ".exe" format file. Default is CP/M-86 ".cmd" format.

-o* write the output to the file *. If -exe is specified, default is "xeq.exe"; otherwise the default is "xeq.cmd".

-s for a CP/M-86 ".cmd" file, make code group shareable, if small model is used.

-zb generate zeros for the bss segment. Default is to reserve space beyond the end of the data group image for bss, as well as for stack and data area growth. The standard startup routines zero the bss segment before _main is called, so this option is needed only for alternate startup sequences.

-zh generate zeros for the stack plus heap, as well as the bss segment. Once again, this should not be necessary, given the standard startup routines.

If <file> is present, it is used as the input rather than the default "xeq".

An output ".cmd" file consists of a 128-byte header, followed by one or two group images. If the data bias in the object file lies below the end of the text segment, then the "small" model is used and both code and data groups are generated. Otherwise, the "8080" model is used and only a combined code/data group is generated. In either case, the group into which data is generated must have no input segment biased below 0x100, to leave room for the base page.

For each group generated, a nine-byte group descriptor is written into the header, commencing with byte zero; the unused portion of the header is filled with zeros. A group descriptor consists of a format byte, followed by four two-byte unsigned paragraph lengths, written less significant byte first. The format byte is 0x01 for a nonsharable code group, 0x02 for a data group, and 0x09 for a sharable code group. The first paragraph length is the size of the corresponding group image in the file, in multiples of sixteen bytes, naturally. The second provides an absolute load address if nonzero; to86 always sets this to zero. The third paragraph length is the minimum space to reserve for the group, which is the length of the group plus (for the group containing data) any area specified for bss and stack plus heap. And the last length provides a maximum space to reserve; by default it is zero, though it may be filled in by -[cgx# dgx#].

The group images that follow are padded as necessary to be a multiple of sixteen bytes in length.

An output ".exe" file consists of a 32-byte header, followed by the group images as described above. The header consists of sixteen two-byte words, each written less significant byte first. Word zero is the magic number 0x5a4d; word one is the number of valid bytes in the last page of the file; word two is the size of the file in 512-byte pages; word four is the size of the header in 16-byte paragraph units, always 32; word five is the size of the bss plus stack and heap to reserve beyond the end of the image, also in paragraph units; word six is the high/low loader switch, always -1 for low memory loading; word seven is the stack pointer; word eight is the stack segment register, relative to the start of the program image; word nine is the checksum of the entire file, summed as an array of two-byte words, the sum then complemented; word ten is the program counter; word 11 is the code segment register, relative to the start of the program image; and word 12 is the offset in the header of the first (non-existent) relocation item, always 28. All other words in the header are zero.

In the small model, the stack segment register points at the start of the combined data/stack group, which follows the code group. In all cases, the code segment register points at the start of the Program Segment Prefix; hence the text segment must be biased at 0x100 for either the 8080 or the small model.

EXAMPLE

To generate a CP/M executable file for the 8080 model:

```
% link -tb0x100 -ed__edata -eb__memory chdr.o PROG.o libc.86
% to86 -b1 -o PROG.cmd
```

And for the small model:

```
% link -db0x100 -ed__edata -eb__memory chdr.o PROG.o libc.86
% to86 -b0x4000 -o PROG.cmd
```

In either case, the standard object file is left at "xeq".

To make a DOS executable file for the 8080 model:

```
% link -tb0x100 -ed__edata -eb__memory chdr.o PROG.o libc.86
% to86 -exe -b0x4000 -o PROG.exe
```

And for the small model:

```
% link -tb0x100 -db0 -ed__edata -eb__memory chdr.o PROG.o libc.86
% to86 -exe -b0x4000 -o PROG.exe
```

Note that a DOS ".com" file can be produced, for the 8080 model, directly by link:

```
% link -htr -tb0x100 -ed__edata -eb__memory -o PROG.com -i
chdr.o
PROG.o
libc.86
```

SEE ALSO

as.86(II), doshdr, link(II)

BUGS

Under DOS, the runtime startup module doshdr.o sets the ds register equal to the ss register before transferring control to the user program. If doshdr.o is not used in a given program, this assignment must still be made before ds is referenced.

Under CP/M, the "maximum group size" field in any group headers output is set to zero by default, purely because this demonstrably works for both 8080 and small model files under CP/M and MP/M. Caveat programmer.

NAME

exit - terminate program execution

SYNOPSIS

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

FUNCTION

exit calls all functions registered with onexit, closes all files, and terminates program execution. If (success == NO) the VERSAdos abort trap is called with a code of 0x8000, otherwise there is normal (quiet) termination.

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 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, i.e., the file is not a teletype.

RETURNS

lseek returns the file descriptor if successful, or a negative number, which is -1 or the VERSAdos return value, negated.

EXAMPLE

To read a 512-byte block:

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

BUGS

It doesn't check for illegal values of offset. If an existing file is opened, the length of the file is taken as the full space allocated for the file, no matter how little has actually been used.

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

BUGS

The type declarations defy description, and are still wrong.

NAME

open - open a file

SYNOPSIS

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

FUNCTION

open opens a file of specified name and assigns a file descriptor to it. If (rsize == 0) carriage returns and NULs are deleted on input, and a carriage return is inserted before each linefeed (newline) on output; if (rsize & 01) the record length is set to 256; else the record length is rsize. mode is ignored.

RETURNS

open returns a file descriptor for the opened file or a negative number, which is -1 or the VERSAdos return value, 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 STDIN is read, and it has not been redirected from the console, characters are read up to and including the first carriage return or newline; carriage return is changed to newline. Otherwise if the file has been opened with (rsize == 0) carriage returns and NULs are deleted; otherwise characters are input unchanged.

RETURNS

If an error occurs, read returns a negative number which is -1 or the VERSAdos return value, 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

open, write

NAME

remove - remove a file

SYNOPSIS

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

FUNCTION

remove deletes the file fname from the filesystem.

RETURNS

remove returns zero, if successful, or a negative number, which is -1 or the VERSAdos error return code, negated.

EXAMPLE

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

NAME

sbreak - set system break

SYNOPSIS

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

FUNCTION

sbreak adjusts the top of the user area, algebraically up by size bytes, rounded up if size is odd.

RETURNS

If successful, sbreak returns a pointer to the start of the added data area, rounded up to a word boundary if necessary; otherwise the value returned is NULL.

EXAMPLE

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

NAME

trap1 - make a task management system call

SYNOPSIS

```
UTINY _trap1(a0arg, d0arg)
    LONG a0arg, d0arg;
```

FUNCTION

_trap1 performs a VERSAdos system call, to perform some task management function. The arguments vary with the actual VERSAdos function desired; a0arg is the value placed in a0 before the call, and d0arg is placed in d0. See chapter 3 of the VERSAdos Preliminary Description for details on these features.

RETURNS

trap1 returns the status code byte returned by the trap. In general, a value of 0 signals success; other codes are function dependent.

SEE ALSO

trap2, trap3

NAME

trap2 - make an input/output system call

SYNOPSIS

```
UTINY _trap2(pios)
    IOS *pios;
```

FUNCTION

_trap2 performs a VERSAdos system call, to perform some input/output function. pios points at a VERSAdos I/O service block, as declared in vdos.h; its contents specify the operation to be performed. See chapter 3 of the VERSAdos Preliminary Description for details on these features.

RETURNS

trap2 returns the status code byte returned by the trap. In general, a value of 0 signals success; other codes are function dependent.

SEE ALSO

trap1, trap3

NAME

trap3 - make a file service system call

SYNOPSIS

```
UTINY _trap3(pfhs)
    FHS *pfhs;
```

FUNCTION

_trap3 performs a VERSAdos system call, to perform some file service function. pfhs points at a VERSAdos file services block, as declared in vdos.h; its contents specify the operation to be performed. See chapter 3 of the VERSAdos Preliminary Description for details on these features.

RETURNS

trap3 returns the status code byte returned by the trap. In general, a value of 0 signals success; other codes are function dependent.

SEE ALSO

trap1, trap2

NAME

uname - create a unique filename

SYNOPSIS

```
TEXT *uname()
```

FUNCTION

uname returns a pointer to the start of a NUL terminated name which is likely not to conflict with normal user filenames. The name may be modified by a one letter suffix. 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.x". 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

create, open, remove

NAME

write - write to a file

SYNOPSIS

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

FUNCTION

write writes size characters starting at buf to the file specified by fd. If STDOUT or STDERR is written, and it has not been redirected from the console, or if the file has been created or opened with (rsize == 0) each newline is preceded by a carriage return. Otherwise characters are output unchanged.

RETURNS

If an error occurs, write returns a negative number which is -1 or the VERSAdos return value, 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

create, open, read

III.c. CP/M-68k System Interface Library

NAME

Interface - to CP/M-68k system

FUNCTION

Programs written in C for operation on the MC68000 under CP/M-68k are translated according to the following specifications:

external identifiers - may be written in both upper and lowercase. The first eight letters must be distinct. An underscore '_' is prepended to each identifier.

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

literal data - such as strings, double constants, and switch tables, are normally generated into .text.

initialized data - is normally generated into .data. External data names are published via .globl declarations.

uninitialized declarations - result in an .globl reference, one instance per program file.

function calls - are performed by

1) moving arguments onto the stack, right to left. Character and short data are widened to long integer, float is converted to double.

2) calling via "jsr _func".

3) popping the arguments off the stack.

Except for returned value, the registers d0, d1, d2, d6, d7, a0, a1, a2, and the condition codes are undefined on return from a function call. All other registers are preserved. The returned value is in d7 (char or short widened to long integer, long integer, and pointer to), or in d6-d7 (float widened to double, and double).

stack frames - are maintained by each C function, using a6 as a frame pointer. On entry to a function, the instruction 'link a6,#' will stack a6 leaving a6 pointing to the stacked a6, and will reserve # bytes for automatics. If more than 32,768 bytes of autos are specified, additional code is generated to reserve space on the stack. Any non-volatile registers used (d3, d4, d5, a3, a4, a5) are then stacked and, if the top of stack scratch cell is used, an additional four bytes are reserved (by either stacking d0 along with other registers or by modifying the stack pointer). Arguments are now at 8(a6), 12(a6), etc. and auto storage is on the stack at -1(a6) on down. To return, the nonvolatile registers are restored from the stack, a6 and a7 are retrieved with an "unlk a6" and control is returned via "rts". Note that the stack must be balanced on exit if the nonvolatile registers are to be properly restored.

data representation - short integers are stored as two bytes, more significant byte first. Long integers are stored as four bytes, in descending order of significance. Floating numbers are represented as for the proposed IEEE Floating Point Standard, four bytes for float, eight for double, and are stored in descending order of byte significance. The IEEE representations are: most significant bit is one for negative numbers, else zero; next eleven bits (eight for float) are the characteristic, biased such that the binary exponent of the number is the characteristic minus 1022 (126 for float); remaining bits are the fraction, starting with the 1/4 weighted bit. If the characteristic is zero, the entire number is taken as zero and should be all zero to avoid confusing some routines that take shortcuts. Otherwise there is an assumed 1/2 added to all fractions to put them in the interval [0.5, 1.0). The value of the number is the fraction, times -1 if the sign bit is set, times 2 raised to the exponent.

storage bounds - even byte storage boundaries must be enforced for multibyte data. 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.

NAME

Conventions - CP/M system subroutines

SYNOPSIS

```
#include <cpm.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 system header file, <cpm.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:

```
CTRLZ - 032, ctl-Z for text end of file
EOF - 1, end of file from CP/M read
FAIL - -1, standard failure return code
MCREATE - 0, copen modes
MOPEN - 1
MREMOVE - 2
MWRITE - 4
SYSBUF - 0x80, location of CP/M buffer
CRESET - 0, CP/M system call codes
CRDCON - 1
CWRCON - 2
CRDRDR - 3
CWRPUN - 4
CWRLST - 5
CDCIO - 6
CGIOST - 7
CSIOST - 8
CPRBUF - 9
CRDBUF - 10
CICRDY - 11
CLFTHD - 12
CINIT - 13
CLOGIN - 14
COPEN - 15
CCLOSE - 16
CSRCH - 17
CSRCHN - 18
CDEL - 19
CREAD - 20
CWRITE - 21
CMAKE - 22
CRENAME - 23
CILOGIN - 24
CIDRNO - 25
CSETAD - 26
CIALLOC - 27
CWPROT - 28
CGETVEC - 29
CSETFA - 30
```

CGETPAR - 31
CGSUSER - 32
CRREAD - 33
CRWRITE - 34
CFSIZE - 35
CSETRR - 36
CDRESET - 37
CRZWRIT - 40
WOPEN - 1, the WCB flags
WDIRT - 2
WSTD - 4
WCR - 010
WUSED - 020
LST - -4, the device codes
PUN - -3
RDR - -2
CON - -1

SEE ALSO

CP/M or CDOS Manual

NAME

c - compiling C programs

SYNOPSIS

A:submit c sfile

FUNCTION

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

The C source file is at sfile.c, where sfile is the name typed in the submit line. The relocatable image from the assembler is put at sfile.o.

EXAMPLE

To compile test.c:

```
A:submit c test
```

SEE ALSO

ld, pc

FILES

sfile.tm?

BUGS

There should be some way to pass the -m flag to p1, or some of the myriad flags acceptable to pp, other than by modifying the command file proper.

NAME

pc - compiling Pascal programs

SYNOPSIS

A:submit pc sfile

FUNCTION

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

The Pascal source file is at sfile.p, where sfile is the name typed in the submit line. The relocatable image from the assembler is put at sfile.o.

EXAMPLE

To compile test.p:

```
A:submit pc test
```

SEE ALSO

c, ld

FILES

sfile.tm?

BUGS

There should be some way to pass flags to ptc, other than by modifying the command file proper.

NAME

ld - linking a C program

SYNOPSIS

A:submit ld ofile

FUNCTION

Programs written in C must be linked with certain object modules that implement the standard C runtime environment. The ld command script invokes the link program, including the standard object header file, the object file ofile, and any libraries in the correct order, and produces the executable image xeq.com.

The standard object header gets control when xeq.com is run. It calls the function _main(), described in the system library, which reads in a command line, redirects the standard input and output as necessary, calls the user provided main(ac, av) with the command arguments, and passes the value returned by main on to exit(). All of this malarkey can be circumvented if file contains a defining instance of _main().

ofile must be a standard object file produced by the assembler, or by an earlier (partial binding) invocation of link. The assembler accepts either the output of the C code generator p2 or assembler source that satisfies the interface requirements of C, as described earlier under Interface. 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:

```
A:submit c echo
A:submit ld echo
A:xeq hello world!
hello world!
```

SEE ALSO

c, pc

NAME

to68k - convert to CP/M-68k executable format

SYNOPSIS

to68k -[bb## o* r t] <file>

FUNCTION

to68k translates standard MC68000 object files to CP/M-68k ".68k" format or CP/M-68k ".rel" format.

The flags are:

-bb## assume the bss bias is set to ##, instead of just beyond the data segment. This flag, if used, must match the "-bb##" flag given to link when the input file was created, since the standard header has no provision for recording a peculiar bss bias.

-o* write the output to the file *. Default is "xeq.68k".

-r suppress relocation bits in the output module.

-t suppress symbol table in the output module.

If <file> is present, it is used as the input rather than the default "xeq".

The output file consists of a 28 or 36 byte header, followed by a text segment, a data segment, the symbol table, and relocation information. The header consists of a short magic number followed by the long number of bytes of object code defined by the text segment, the long number of bytes defined by the data segment, the long number of bytes defined by the bss segment, the long number of bytes in the symbol table, a zero padding long, the long relocation bias of the text segment, and a short that is nonzero if relocation bits are suppressed. To this may be appended the long relocation bias of the data segment and the long relocation bias of the bss segment. All integers in the object image are written most significant byte first, in descending order of significance.

The value of the magic number is determined by the relationship between the text, data, and bss biases in the input file header. If the data segment immediately follows the text, and the bss segment immediately follows the data, the magic number is 0x601a and the shorter header is used. Otherwise the magic number is 0x601b and the longer header is used to specify the additional biases.

Text and data segments each consist of an integral number of shorts.

Each symbol table entry consists of an eight-byte name padded with trailing NULs, a flag short, and a value long. Meaningful flag values are 0x200 for text relative, 0x400 for data relative, and 0x100 for bss relative. To this is added 0x8000 if the symbol is defined, and 0x2000 if the symbol is to be globally known.

One short of relocation information is present for each short of text and data. A relocation short of 0 implies no relocation, 02 is for text relative, 01 for data relative, 03 for bss relative, and 04 for an undefined external symbol whose index into the symbol table is the relocation word right shifted three. A short of 05 flags the upper half of a long to be relocated by the short following, and a short of 06 implies 16-bit PC-relative relocation.

EXAMPLE

To translate a standard object file to MC68000 ".68k" format:

```
% link -tb0x1200 -ed__edata -eb__memory Crtcpm.68k PROG.o libccpm.68k
% to68k -o PROG.68k
```

Note that a ".68k" file can be produced, without relocation bits or symbol table, directly by link, by using the files cout or coutl, which emulate a CP/M-68k executable file header. The following command line would produce a file biased to run at location 0x1200, with a magic number of 0x601a:

```
% link -tb0x11e4 -htr -ed__edata -eb__memory -st__stext \
    -sd__sdata -sb__sbss -o PROG.68k -i
cout.68k
Crtcpm.68k
PROG.o
libccpm.68k
```

To translate a standard object file to MC68000 ".rel" format:

```
% link -ed__edata -eb__memory Crtcpm.68k PROG.o libccpm.68k
% to68k -o PROG.rel
```

This is how to68k should be used in conjunction with the standard compiler driver scripts; the link line shown is equivalent to the one in the scripts. A ".rel" file can be converted to ".68k" format under CP/M-68k.

SEE ALSO

as.68k(II), cout, coutl, link(II)

NAME

chdr - C runtime entry

SYNOPSIS

```
link -tb0x1200 -ed__edata -eb__memory a:chdr.o <files> libc.68k
to68k -o prog.68k
```

FUNCTION

All CP/M-68k programs begin execution at location 0x100 in the program base; chdr.o is the startup routine, linked at this address, that sets the stack to the address stored at location 6 in the program base, clears bss (by zeroing memory from __edata to __memory), then calls _main().

Included in the chdr module is the C callable interface function cpm(), plus several internal functions that isolate 68000 machine dependencies.

SEE ALSO

_main, cpm

NAME

cout - simulate a CP/M-68k executable file header

SYNOPSIS

```
link -eb__memory -ed__edata -st__stext -sd__sdata -sb__sbss -htr \
    -tb-28 cout.68k Crtcpm.68k <files> libucpm.68k libccpm.68k
```

FUNCTION

cout is used in conjunction with the stadard link utility to generate ".68k" files that will execute under CP/M on an MC68000. The text bias (value for -tb) given to link should be the actual runtime location desired minus 28 bytes. cout will cause a ".68k" file with a magic number of 601A to be generated.

EXAMPLE

To produce a ".68k" file, without relocation bits or symbol table, linked to run at location 0x500:

```
% link -eb__memory -ed__edata -st__stext -sd__sdata -sb__sbss -htr \
    -tb0x4e4 cout.68k Crtcpm.68k <files> libucpm.68k libccpm.68k
```

NAME

coutl - simulate a long CP/M-68k executable file header

SYNOPSIS

```
link -eb__memory -ed__edata -st__stext -sd__sdata -sb__sbss -htr \
    -tb-36 coutl.68k Crtcpm.68k <files> libucpm.68k libccpm.68k
```

FUNCTION

coutl is used in conjunction with the standard link utility to generate ".68k" files that will execute under CP/M on an MC68000. The text bias (value for -tb) given to link should be the actual runtime location desired minus 36 bytes. coutl will cause a ".68k" file with a magic number of 601B to be generated.

EXAMPLE

To produce a ".68k" file, without relocation bits or symbol table, linked to run at location 0x500, with a data bias rounded up to the next higher 64-byte boundary:

```
% link -eb__memory -ed__edata -st__stext -sd__sdata -sb__sbss -htr \
    -tb0x4dc -dr6 coutl.68k Crtcpm.68k <files> libucpm.68k libccpm.68k
```

NAME

_main - setup for main call

SYNOPSIS

```
BOOL _main()
```

FUNCTION

_main is the function called whenever a C program is started. It parses the command line at 0x80 (in the program base) 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 spaces. 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 taken from _pname.

Note that the argument strings remain in the default record buffer beginning at 0x80 in the program base, which is never used by other C interface routines.

EXAMPLE

To avoid the loading of _main and all the 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, cpm, exit

NAME

_pname - program name

SYNOPSIS

```
TEXT *_pname;
```

FUNCTION

_pname is the (NUL terminated) name by which the program was invoked, at least as anticipated at compile time. If the user program provides no definition for _pname, a library routine supplies the name "error", since it is used primarily for labelling diagnostic printouts.

Argument zero of the command line is set equal to _pname.

SEE ALSO

_main

NAME

close - close a file

SYNOPSIS

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

FUNCTION

close closes the file associated with the file descriptor fd, making the fd available for future open or create calls. If the file was written to or created, close ensures that the last record is written out and the directory entry properly closed.

RETURNS

close returns the now useless file descriptor, if successful, or -1.

EXAMPLE

To copy an arbitrary number of files:

```
while (0 <= (fd = getfiles(&ac, &av, STDIN, -1)))
    {
    while (0 < (n = read(fd, buf, BUFSIZE)))
        write(STDOUT, buf, n);
    close(fd);
    }
```

SEE ALSO

create, open, remove, uname

NAME

cpm - call CP/M-68k system

SYNOPSIS

```
COUNT cpm(d0, d1)
    COUNT d0;
    TEXT *d1;
```

FUNCTION

cpm is the C callable function that permits arbitrary calls to be made on CP/M-68k. It loads its arguments into registers d0 and d1, then performs a system call via trap #2. The function to be performed is specified by the less significant word of d0, i.e. d0.w; typically d1 contains a word integer or a pointer.

RETURNS

cpm returns the d0.b register returned by the CP/M-68k call, sign extended to a long integer.

EXAMPLE

To read a line from the console:

```
buf[0] = 125;
cpm(CRDBUF, buf);
cpm(CWRCON, '\n');
```

BUGS

The return value from the system call may be corrupted, if larger than a signed character.

NAME

create - open an empty instance of a file

SYNOPSIS

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

FUNCTION

create makes a new file of specified name, 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). This mode information is largely ignored, however.

If (rsize is zero), carriage returns and NULs are deleted on input, a ctl-Z is treated as an end of file, a carriage return is injected before each newline on output, and a ctl-Z is appended to the data in a partially filled last record. If (rsize != 0) data is transmitted unaltered.

Filenames take the general form x:name.typ, where x is the disk designator, name is the eight-character filename, and typ its three-character type. If x: is omitted, it is taken as the disk logged in on the first call to create, open, or remove; a missing name or typ is taken as all blanks. Letters are forced uppercase.

The four physical devices con:, rdr:, pun:, and lst: are also accepted as filenames. Reading from pun: or lst: gives an instant end of file, while writing to rdr: sends all bytes to hell with no complaint.

RETURNS

create returns a file descriptor for the created file or -1.

EXAMPLE

```
if ((fd = create("xeq", WRITES, 1)) < 0)
    write(STDERR, "can't create xeq\n", 17);
```

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.

RETURNS
exit will never return to the caller.

EXAMPLE
```
if ((fd = open("file", READ)) < 0)
    {
    write(STDERR, "can't open file\n", 16);
    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 system uses only the low order 31 bits of the offset; the sign is ignored.

RETURNS

lseek returns zero if successful, or -1.

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);
    }
```

BUGS

The length of the file is taken as 128 times the total number of records, even if a text file is terminated early with a ctl-Z.

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

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 keyboard interrupt, usually caused by a DEL key typed during terminal input or output. (Under DOS on the 8086, a CTL-BREAK also serves this function.) 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, an immediate error exit is taken.

If (pfn == NULL) then these interrupts are disabled (turned off).

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 an existing file

SYNOPSIS

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

FUNCTION

open associates a file descriptor with an existing file. 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). This mode information is largely ignored, however.

If (rsize == zero), carriage returns and NULs are deleted on input, a ctl-Z is treated as an end of file, a carriage return is injected before each newline on output, and a ctl-Z is appended to the data in a partially filled last record. If (rsize != 0) data is transmitted unaltered.

Filenames take the general form x:name.typ, where x is the disk designator, name is the eight-character filename, and typ its three-character type. If x: is omitted, it is taken as the disk logged in on the first call to create, open, or remove; a missing name or typ is taken as all blanks. Letters are forced uppercase.

The four physical devices con:, rdr:, pun:, and lst: are also accepted as filenames. Reading from pun: or lst: gives an instant end of file, while writing to rdr: sends all bytes to hell with no complaint.

RETURNS

open returns a file descriptor for the file or -1.

EXAMPLE

```
if ((fd = open("xeq", WRITES, 1)) < 0)
    write(STDERR, "can't open xeq\n", 15);
```

SEE ALSO

close, create, remove, uname

NAME

read - read characters 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) carriage returns and NULs are discarded on input.

Reading from pun: or lst: always gives a count of zero.

If the console is read, DEL at the start of a line causes an interrupt, to be processed as specified by onintr.

RETURNS

If an error occurs, read returns -1; 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

onintr, write

NAME
remove - remove a file

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

FUNCTION
remove deletes the file fname from the filesystem.

RETURNS
remove returns zero, if successful, or -1.

EXAMPLE
```
if (remove("temp.c") < 0)
    write(STDERR, "can't remove temp file\n", 23);
```

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.

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))))
    {
    write(STDERR, "not enough room!\n", 17);
    exit(NO);
    }
```

BUGS

The stack is assumed to lie above the data area, so sbreak will return a NULL if the new system break lies above the current stack pointer; this may not be desirable behavior on all memory layouts.

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 likely not to conflict with normal user filenames. The name may be modified by a letter 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, which is currently the string "ctempc.". The pointer will never be NULL.

EXAMPLE

```
if ((fd = create(uname(), WRITE, 0)) < 0)
    write(STDERR, "can't create sort temp\n", 23);
```

SEE ALSO

close, create, open, remove

NAME

write - write characters to a file

SYNOPSIS

```
COUNT write(fd, buf, size)
    FILE fd;
    TEXT *buf;
    COUNT 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), each newline output is preceded by a carriage return. Moreover, a ctl-Z is appended to a file that does not end on a record boundary.

RETURNS

If an error occurs, write returns -1; otherwise the value returned should be size. Writing to rdr: does nothing, but returns size as if it did everything.

Characters typed at the console are inspected during write calls; typing a DEL causes an interrupt, to be processed as specified by onintr.

EXAMPLE

To copy a file:

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

SEE ALSO

onintr, read

IV. Machine Support Library for MC68000

NAME

Conventions - the MC68000 Machine Support Library

FUNCTION

The modules in this section are called upon by code produced by the MC68000 C compiler, primarily to perform operations too lengthy to be expanded in-line. Functions are described in terms of their assembler interface, since they operate outside the conventional C calling protocol.

Unless explicitly stated otherwise, every function does obey the normal C calling convention that the condition code and registers d0, d1, d2, d6, d7, a0, a1, and a2 are not preserved across a call.

NAME

check - check stack on entering a C function

SYNOPSIS

```
move.l  $-nautos,d0
jsr     a.check
```

FUNCTION

a.check ensures that d0+sp is not lower than _stop, to check for stack overflow.

If stack overflow would occur, the _memerr condition is raised. This will never happen if _stop is set to zero.

RETURNS

Nothing.

EXAMPLE

The C function:

```
COUNT idiot()
    {
    COUNT i;

    return (i);
    }
```

can be written:

```
idiot:
    move.l  #-24,d0
    jsr     a.check
    link    a6,#-2
    move.w  -2(a6),d7
    ext.l   d7
    unlk    a6
    rts
```

SEE ALSO

_stop

NAME

count - counter for profiler

SYNOPSIS

```
        lea     lbl,a0
        jsr     a.count
```

FUNCTION

a.count is the function called on entry to each function when the compiler is run with the flag "-p" given to p2.68k. lbl is the address of a pointer variable, initially NULL, that is used by a.count to aid in its bookkeeping.

On entry to a.count, the return link is always presumed to be 12 bytes beyond a function entry point.

RETURNS

Nothing.

NAME

dadd - add double into double

SYNOPSIS

```
lea     right,a0    OR  clr.l   a0
jsr     a.1add      OR  jsr     a.6add
```

FUNCTION

a.1add and a.6add are the internal routines called to add the double at right into a double accumulator, called left, which is d1/d2 for a.1add or d6/d7 for a.6add; if a0 is NULL, the other accumulator is taken as the right operand. The addition is performed without destroying any volatile registers, so the call can be used much like an ordinary machine instruction.

If right is zero, left is unchanged (x+0); if left is zero, right is copied into it (0+x). Otherwise the number with the smaller characteristic is shifted right until it aligns with the other and the addition is performed algebraically. The answer is rounded.

RETURNS

dadd replaces its left operand with the closest internal representation to the rounded sum of its operands. Volatile registers and the stack are preserved. The NZ condition codes are set to reflect the result.

SEE ALSO

ddiv, dmul, dsub, repk, unpk

BUGS

It doesn't check for characteristics differing by huge amounts, to save shifting. If the right operand is zero, an unnormalized left operand is left unchanged.

NAME

in - input byte from port

SYNOPSIS

```
COUNT in(port)
    UCOUNT port;
```

FUNCTION

in is a C callable function to input a byte from an arbitrary port, using the instruction "in al,dx" with port in dx.

RETURNS

in returns the byte read as an integer whose more significant byte is zero.

EXAMPLE

```
while (!(in(STATPORT) & DONE))
    ;
out(DATAPORT, ch);
```

SEE ALSO

inw, out, outw

NAME

inw - input word from port

SYNOPSIS

```
COUNT inw(port)
    UCOUNT port;
```

FUNCTION

inw is a C callable function to input a word from an arbitrary port, using the instruction "in ax,dx" with port in dx.

RETURNS

inw returns the word read as an integer.

EXAMPLE

```
while (!(inw(STATPORT) & DONE))
    ;
outw(DATAPORT, ch);
```

SEE ALSO

in, out, outw

NAME

dmul - multiply double into double

SYNOPSIS

```
lea     right,a0    OR  clr.l   a0
jsr     a.1mul      OR  jsr     a.6mul
```

FUNCTION

a.1mul and a.6mul are the internal routines called to multiply the double at right into a double accumulator, called left, which is d1/d2 for a.1add or d6/d7 for a.6add; if a0 is NULL, the other accumulator is taken as the right operand. The multiplication is performed without destroying any volatile registers, so the call can be used much like an ordinary machine instruction.

If either right or left is zero, the result is zero (0*x, x*0). Otherwise the right fraction is multiplied into the left and the right exponent is added to that of the left. The sign of a non-zero result is negative if the left and right signs differ, else it is positive. The result is rounded.

RETURNS

dmul replaces its left operand with the closest internal representation to the rounded product of its operands. Volatile registers and the stack are preserved. The NZ condition codes are set to reflect the result.

SEE ALSO

dadd, ddiv, dsub, repk, unpk

BUGS

If the left operand is an unnormalized zero, it is left unchanged.

NAME
out - output byte to port

SYNOPSIS
```
COUNT out(port, data)
    UCOUNT port, data;
```

FUNCTION
out is a C callable function to output a byte to an arbitrary port, using the instruction "out dx,al" with port in dx and data in ax.

RETURNS
out returns data.

EXAMPLE
```
while (!(in(STATPORT) & DONE))
    ;
out(DATAPORT, ch);
```

SEE ALSO
in, inw, outw

NAME
outw - output word to port

SYNOPSIS
```
COUNT outw(port, data)
    UCOUNT port, data;
```

FUNCTION
outw is a C callable function to output a word to an arbitrary port, using the instruction "out dx,ax" with port in dx and data in ax.

RETURNS
outw returns data.

EXAMPLE
```
while (!(inw(STATPORT) & DONE))
    ;
outw(DATAPORT, ch);
```

SEE ALSO
in, inw, out

NAME

dtl - convert double to long

SYNOPSIS

```
jsr     a.1dtl      OR  jsr     a.6dtl
```

FUNCTION

a.1dtl and a.6dtl are the internal routines called to convert the double accumulator, which is d1/d2 for a.1dtl or d6/d7 for a.6dtl, to a long integer in d0. They do so by calling unpk, to separate the fraction from the characteristic, then shifting the fraction until the binary point is at a known fixed place. The long integer immediately to the left of the binary point is delivered, with the same sign as the original double. Truncation occurs toward zero.

RETURNS

dtl returns a long in d0, which is the low-order 32 bits of the integer representation of the double in the accumulator used, truncated toward zero. Volatile registers and the stack are preserved.

SEE ALSO

ltd, ultd

NAME

ftd - convert float to double

SYNOPSIS

```
jsr     a.1ftd      OR  jsr     a.6ftd
```

FUNCTION

a.1ftd and a.6ftd are the internal routines called to convert a float in the more significant half of a double accumulator, which is d1/d2 for a.1ftd or d6/d7 for a.6ftd, to a double in the accumulator.

RETURNS

ftd returns a double in the accumulator used. The NZ condition codes are set to reflect the result. Volatile registers and the stack are preserved.

SEE ALSO

dtf

NAME

ldiv - divide long by long

SYNOPSIS

```
move.l  n,-(sp)
move.l  d,-(sp)
jsr     a.ldiv
```

FUNCTION

a.ldiv divides the long n by the long d to obtain the long quotient. The sign of a non-zero result is negative only if the signs of n and d differ. No check is made for division by zero, which currently gives a quotient of -1 or +1.

RETURNS

The value returned is the long quotient of n / d on the stack. Volatile registers are preserved. The NZ condition codes are set to reflect the result.

SEE ALSO

lmod, lmul, uldiv, ulmod

NAME

lmod - divide long by long and return remainder

SYNOPSIS

```
move.l  n,-(sp)
move.l  d,-(sp)
jsr     a.lmod
```

FUNCTION

a.lmod divides the long n by the long d to obtain the long remainder. The sign of a non-zero result is negative only if the sign of n is negative. No check is made for division by zero, which currently gives a remainder equal to the value of n.

RETURNS

The value returned is the long remainder of n / d on the stack. Volatile registers are preserved. The NZ condition codes are set to reflect the result.

SEE ALSO

ldiv, lmul, uldiv, ulmod

NAME

lmul - multiply long by long

SYNOPSIS

```
move.l  l,-(sp)
move.l  r,-(sp)
jsr     a.lmul
```

FUNCTION

a.lmul multiplies the long l by the long r to obtain the long product. The sign of a non-zero result is negative only if the signs of l and r differ. No check is made for overflow, which currently gives the low order 32 bits of the correct product.

RETURNS

The value returned is the long product l * r on the stack. Volatile registers are preserved. The NZ condition codes are set to reflect the result.

SEE ALSO

ldiv, lmod, uldiv, ulmod

NAME

ltd - convert long to double

SYNOPSIS

```
move.l  n,d0
jsr     a.1ltd      OR  jsr     a.6ltd
```

FUNCTION

a.1ltd and a.6ltd are the internal routines called to convert the long integer in d0 to a double in an accumulator, which is d1/d2 for a.1ltd or d6/d7 for a.6ltd. They do so by extending the long to an unpacked double fraction, then calling repk with a suitable characteristic. The conversion is performed without destroying any volatile registers, so the call can be used much like an ordinary machine instruction.

RETURNS

ltd writes the double equivalent of the long in d0 into the specified accumulator. Volatile registers and the stack are preserved. The NZ condition codes are set to reflect the result.

SEE ALSO

dtl, repk, ultd

NAME

ltf - convert long to float

SYNOPSIS

```
move.l  n,d0
jsr     a.ltf
```

FUNCTION

a.ltf is the internal routine called to convert the long integer in d0 to a float in d0. It does so by calling ltd and dtf in turn, and without destroying any volatile registers, so that the call can be used much like an ordinary machine instruction.

RETURNS

ltf writes the float equivalent of the long in d0 into d0. Volatile registers and the stack are preserved. The NZ condition codes are set to reflect the result.

SEE ALSO

dtf, ltd, ultf

NAME

repk - repack a double number

SYNOPSIS

```
move.l  char,-(sp)
pea     frac
jsr     a.repk
addq.l  #8,sp
```

FUNCTION

a.repk is the internal routine called by various floating runtime routines to pack a signed fraction at frac and binary characteristic char into a standard form double representation. The fraction occupies two four-byte integers, starting at frac, and may contain any value; there is an assumed binary point immediately to the right of the most significant byte. The characteristic is 1023 plus the power of two by which the fraction must be multiplied to give the proper value.

If the fraction is zero, the resulting double is all zeros. Otherwise the fraction is forced positive and shifted left or right as needed until the most significant bit is at the 020 position in byte #1 (counting from zero), with the characteristic being incremented or decremented as appropriate. The fraction is then rounded to 53 binary places. If the resultant characteristic can be properly represented in a double, it is put in place and the sign is set to match the original fraction sign. If the characteristic is zero or negative, the double is all zeros. Otherwise the characteristic is too large, so the double is set to the largest representable number, and is given the sign of the original fraction.

RETURNS

repk replaces the two four-byte integers of the fraction with the double representation. The value of the function is VOID, i.e., garbage. The NZ condition codes are set to reflect the result, however, which is nonstandard.

SEE ALSO

unpk

BUGS

Really large magnitude values of char might overflow during normalization and give the wrong approximation to an out of range double value.

NAME

switch - perform C switch statement

SYNOPSIS

```
        move.l  val,d7
        lea     swtab,a0
        jmp     a.switch
```

FUNCTION

a.switch is the code that branches to the appropriate case in a switch statement. It compares val against each entry in swtab until it finds an entry with a matching case value or until it encounters a default entry. swtab entries consist of zero or more (lbl, value) pairs, where lbl is the (nonzero) address to jump to and value is the case value that must match val.

A default entry is signalled by the pair (0, deflbl), where deflbl is the address to jump to if none of the case values match. The compiler always provides a default entry, which points to the statement following the switch if there is no explicit default statement within the switch.

RETURNS

switch exits to the appropriate case or default; it never returns.

NAME

uldiv - divide unsigned long by unsigned long

SYNOPSIS

```
move.l  n,-(sp)
move.l  d,-(sp)
jsr     a.uldiv
```

FUNCTION

a.uldiv divides the long n by the long d to obtain the unsigned long quotient. No check is made for division by zero, which currently gives a quotient of -1.

RETURNS

The value returned is the long unsigned quotient of n / d on the stack. Volatile registers are preserved. The NZ condition codes are set to reflect the result.

SEE ALSO

ldiv, lmod, lmul, ulmod

NAME

ulmod - unsigned long remainder

SYNOPSIS

```
move.l  n,-(sp)
move.l  d,-(sp)
jsr     a.ulmod
```

FUNCTION

a.ulmod divides the long n by the long d to obtain the unsigned long remainder. No check is made for division by zero, which currently gives a remainder of n.

RETURNS

The value returned is the long unsigned remainder of n / d on the stack. Volatile registers are preserved. The NZ condition codes are set to reflect the result.

SEE ALSO

ldiv, lmod, lmul, uldiv

NAME

ultd - convert unsigned long to double

SYNOPSIS

```
move.l  n,d0
jsr     a.1ultd     OR  jsr     a.6ultd
```

FUNCTION

a.1ultd and a.6ultd are the internal routines called to convert the unsigned long integer in d0 to a double in an accumulator, which is d1/d2 for a.1ultd or d6/d7 for a.6ultd. They do so by extending the long to an unpacked double fraction, then calling repk with a suitable characteristic. The conversion is performed without destroying any volatile registers, so the call can be used much like an ordinary machine instruction.

RETURNS

ultd writes the double equivalent of the unsigned long in d0 into the specified accumulator. Volatile registers and the stack are preserved. The NZ condition codes are set to reflect the result.

SEE ALSO

dtl, ltd, repk

NAME

ultf - convert unsigned long to float

SYNOPSIS

```
move.l  n,d0
jsr     a.ultf
```

FUNCTION

a.ultf is the internal routine called to convert the unsigned long integer in d0 to a float in d0. It does so by calling ultd and dtf in turn, and without destroying any volatile registers, so that the call can be used much like an ordinary machine instruction.

RETURNS

ultf writes the float equivalent of the unsigned long in d0 into d0. Volatile registers and the stack are preserved. The NZ condition codes are set to reflect the result.

SEE ALSO

dtf, ltf, ultd

NAME

unpk - unpack a double number

SYNOPSIS

```
        pea     double
        pea     frac
        jsr     a.unpk
        addq.l  #8,sp
```

FUNCTION

a.unpk is the internal routine called by various floating runtime routines to unpack a double at double into a signed fraction at frac and a characteristic. The fraction consists of two four-byte integers at frac; the binary point is immediately to the right of the most significant byte. If the double at double is not zero, unpk guarantees that the most significant fraction byte is exactly equal to 1, and the least significant fraction byte has three zero bits for use as guard, rounding, and sticky bits.

The characteristic returned is 1023 plus the power of two by which the fraction must be multiplied to give the proper value; it will be zero for any flavor of zero at double, i.e., having a characteristic of zero, irrespective of other bits.

RETURNS

unpk writes the signed fraction as two four-byte integers starting at frac and returns the characteristic in d7 as the value of the function.

SEE ALSO

repk

Whitesmiths, Ltd.
97 Lowell Road, Concord, Massachusetts 01742