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 *W compilers. This enhances uniformity of language implementation and simplifies programming for more than one target machine.

WARNINGS

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 arguments 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.

WARNINGS

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.

WARNINGS

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

NAME

as.370 - assembler for IBM/370

SYNOPSIS

as.370 -[o* x] <files>

FUNCTION

as.370 translates IBM/370 assembly language to standard format relocatable object images. Since the output of the C code generator p2 is assembly code, as.370 is required to produce relocatable images suitable for binding with link.

The flags are:

-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.

-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 no -o flag is present, and one or more <files> are present, and the first filename ends with ".s", then as.370 behaves as if -o was specified using the first filename, only with the trailing 's' changed to 'o'. Thus,

```
as.370 file.s
```

is the same as:

```
as.370 -o file.o file.s
```

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 the the short int 0x996c, 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.

The length of all segments is rounded up to a multiple of eight bytes. 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 reloction of the next short or long integer in the corresponding segment. If the 1-weigthed bit is set int the control byte, then a change in load bias must be subtracted from the integer. If the 2-weigthed bit is set, a long integer rather than a short is to be relocated.

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 word; 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 word.

Each symbol table entry consists of a four-byte value, a flag byte, and an nine-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 for it, should there be no explicit defining reference.

SEE ALSO

link, p2.370, rel

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.

WARNINGS

Needs some way to specify flags to the prototype programs.

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

NAME
hex - translate object file to ASCII formats

SYNOPSIS
hex -[c## db## dr h m* o* r## s s2 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:

-c## output only ## bytes.

-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 gencmd utility under CP/M-86.

-h don't produce start record for Intel hex files, or S0 record for Motorola S-records.

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

-o* write output module to file *. The default is STDOUT.

-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.

-s2 produce only S2-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.

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. If -h is specified, the S0 record is omitted. 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
S11100000020068656C6C6F20776F726C640090
S9030000FC
```

SEE ALSO
link, rel

NAME

lib - maintain libraries

SYNOPSIS

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

Under Idris and Unix, if a filename in the list <files> contains the character '/', the filename actually recorded or searched for in the library is formed from the longest suffix of filename that does not contain a '/'. Under other operating systems, if a filename in the list <files> contains any of the characters ':', ']' or '/', the filename actually recorded or searched for in the library is formed from the longest suffix of filename that does not contain a ':', ']' or '/'.

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 libc -c one.o two.o three.o
% lib libc -tv
```

SEE ALSO

link, lord, rel

WARNINGS

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 result@ing 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 resul-t@ing 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", and outputs a partially-ordered list. 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 +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** (see below), 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 elements are forced to a multiple-of-4 byte boundary. A bound of 3 is the same as 2, except that 8 byte data elements (doubles) are forced to a multiple-of-8 byte boundary.

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

-e don't force loading of external 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 a structure **x**, with **m** as one of its members.

-n# ignore characters after the first # in testing external identifiers for equality. The default is 7. The 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. The 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**.

+u take declarations of **char** to be **unsigned char**. This flag is useful when the underlying character representation is EBCDIC.

If **<file>** is present on the command line, 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

WARNINGS

It can be difficult to predict **p1**'s interpretation of semicolons.

NAME

p2.370 - generate code for IBM/370 C programs

SYNOPSIS

p2.370 -[b# ck l n* o* p sd# sn# st u] <file>

FUNCTION

p2.370 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 IBM/370 assembler. Two versions of **p2.370** are available: one that generates code for IBM/OS Assembler-XF and one that generates code for a standard IDRIS assembler.

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 small set of machine-dependent runtime library routines. The IBM/370 runtime library is documented in section IV of this manual.

The flags are:

-b# use # additional base-registers in each function. For normal-sized functions, and even for quite large ones, a single base-register, BASEREGS=0, is sufficient. Each additional base-register will permit the functions to be 4096 bytes larger. If the assembler complains about addressability errors, you should try increasing the number of registers with this flag. The default value for # is zero; i.e., only one base-register is used.

-ck enable overflow checking.

-l produce NAME and ALIAS statements for use with the Linkage Editor for each external name-definition in the input file. If a module name is given with **-n***, **-l** produces a NAME statement for that name. This flag is ignored in the version of **p2.370** that generates code for Whitesmiths' assembler. Refer to the documentation of the program **tobj** for a decription of how to generate NAME and ALIAS statements. (Note: this flag may only be used with an assembler capable of assembling more than one deck at a time, e.g., Waterloo's Assembler-G.)

-n* use * as the control-section name in the assembler output. Default is that an unnamed CSECT is emitted. See also the note under **-l** above. In the version of **p2.370** that generates code for Whitesmiths' assembler this flag generates ordinary external symbols. Whether a named CSECT will be generated or not is up to the program **tobj**.

-o* write the output to 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. This flag is not yet fully supported.

-sd# use an indexed switch table if the density of the table is as least # %.

-sn# use an indexed switch table if the number of entries is less or equal to #.

-st emit (12 bytes of) code that branches to the C program startup routine **c~start** (**C#START**) at the beginning of the assembler file. This allows the compiled program to be the only explicit input to the linker or loader.

-u take all declarations of **char** to be **unsigned char**. This flag is obsolete. It is replaced by the +u flag to **p1**.

-x# map the three virtual sections, for Functions (04), Literals (02), and Variables, (01) to the two physical sections Code (bit is one) and Data (bit is zero). This flag has effect only when using the version of **p2.370** that produces code for Whitesmiths' assembler.

If **<file>** is present on the command line, it is used as the input file instead of the default STDIN. On many systems (IBM/OS for instance), **<file>** is mandatory, because STDIN is interpreted as a text file, and hence becomes corrupted.

Files output from **p1** for use with the IBM/370 code generator must be generated with the following flags: the **-l** flag, since pointers and **int**s are long; the **-b3** flag; and the **-n7** flag. No **-a** flag should be used since registers are interchangeable.

Wherever possible, labels in the emitted code each contain a comment which gives the source line number from which the code immediately following is obtained.

EXAMPLE

p2 usually follows **pp** and **p1**, as follows:

```
pp -o SYSUT1 -x SYSIN
p1 -o SYSUT2 -l -b3 -n7 SYSUT1
p2.370 -o SYSPUNCH SYSUT2
```

SEE ALSO

pp, p1

NAME

pds - create and extract IBM files in IEBUPDTE-format

SYNOPSIS

pds -[bin bs# c f* i nrw p txt t v x] <files>

FUNCTION

pds administers a IEBUPDTE-format "tape", i.e., a single physical file, written or read as bs-byte blocks, that can represent numerous files transparently. When written to a tape, or other interchange media, a IEBUPDTE file facilitates transfer of multiple files between Idris and IBM/OS systems.

pds is primarily used for text files, but can also be used on binary files if some precautions are taken, as explained under BUGS below.

The name of each file in <files> should be eight characters or less in length. Longer filenames are truncated, with a warning message. Unfortunately pds doesn't complain about multiple occurrences of the same filename. The names are translated in the same way as filenames are translated to ddnames.

When extracting, printing, or tabulating a tape, only the named files participate. By special dispensation, flags of the form "-bin", and "-txt" may be interspersed among <files>, and cause the files named afterwards to be treated as either text or binary. If no <files> are given, all files on the tape are used.

The flags are:

-bin treat the <files> following as binary files.

-bs# specify the blocksize to be used on the tape. Must be a multiple of 80. Default is 32720 if no tape name is specified, otherwise 800 if no blocksize is specified.

-c create a tape. All named <files> are composed into a IEBUPDTE file in the order specified.

-f* use the string * as the name of the tape. Default is /dev/rmt0.

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

-nrw don't rewind tape after use, i.e., use "/dev/rmt8" as tape-file.

-p print files from the tape to STDOUT.

-txt treat the <files> following as text files. This is the default treatment.

-t tabulate the tape.

-v be verbose.

-x extract files from the tape into disk files with the same name.

At most one of -c, -p, -t, or -x may be specified; the default is -t. When -v is specified with -c or -x, the name of each file processed is written to STDOUT. When -v is specified with -t, the number of records occupied by each file is printed together with the name of the file.

RETURNS

pds returns success if all files specified can be processed.

SEE ALSO

Files for ddname translation conventions.

WARNINGS

IEBUPDTE uses records beginning with "./" as delimiters. pds will issue a warning if it encounters the sequence "./" at the start of an output record, but will continue processing.

This problem is particularly vexing when creating a tape containing binary files. The only way around it is to change the composition of the tape, for instance, by reordering the members of an object library.

NAME

pp - preprocess defines and includes

SYNOPSIS

pp -[c d*^ i* map* 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 */ or 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; separate them with the character ¦.

-map* use * as the name of a translation table which is used to translate character constants and strings from one character representation to another. Characters written as octal numbers (e.g. **\007**) are not translated. The size of this file must be at least 256 bytes.

-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 for **ptc**.

pp processes the files named (or STDIN if none are given) in the order specified, writing the resulting text to STDOUT. Preprocessor actions are described in detail in Section II of the C Language 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

WARNINGS

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

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 *W 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.

WARNINGS

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

NAME
tobj - convert IDRIS IBM/370 object to IBM object module

SYNOPSIS
tobj -[c l o* r si t v] <file>

FUNCTION
tobj translates an IDRIS IBM/370 object file to an IBM object card format file.

The flags are:

-c produce a named CSECT as output file. The name is chosen as follows: If the external symbol **_main** is found in the text section, it is used. Otherwise, the first external symbol is used. The first character in this symbol is replaced by a **@**. Thus a file containing a main program will have the CSECT name **@MAIN**.

-l emit NAME and ALIAS statements for all external symbols defined in the file, for use by the linkage editor or the CMS TXTLIB command.

-o* write the output to the file *. Default is **a.obj**.

-r invoke the replace option on the NAME statement (" NAME $PRINTF(R) ") if -l is also used.

-si produce an object module that can be used under SIEMENS BS1000.

-t suppress symbol table in the output module.

-v be verbose.

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

The output file consists of a sequence of object cards, optionally followed by NAME and ALIAS statements.

SEE ALSO
as.370, link