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.11 - assembler for PDP-11

SYNOPSIS

as.11 -[o* x] <files>

FUNCTION

as.11 translates PDP-11 assembly language to standard format relocatable object images. Since the output of the C code generator p2 is assembly code, as.11 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.11 behaves as if -o was specified using the first filename, only with the trailing 's' changed to 'o'. Thus,

```
as.11 file.s
```

is the same as:

```
as.11 -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 byte 0x99, followed by the byte 0x34, followed by seven (two-byte) unsigned words specifying the number of symbol table bytes, 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 zero words, and the data segment offset, which always equals the text segment size. All words in the object image are written less significant byte first.

Text and data segments each consist of an integral number of words. 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 correspon-

ding 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 word 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 word. The 2-weighted bit is always zero for the PDP-11; 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 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 value word, a flag byte, and a 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, should there be no explicit defining reference.

SEE ALSO

link, p2.11, 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.

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.