SECTION ONE

WHITESMITHING

NAME

Process - rules for an Idris program

FUNCTION

Idris provides a relatively clean and simple execution environment. Memory layout is straightforward, input/output is made to look identical for a large variety of devices and files, and system services are packaged as functionally cohesive routines. Programs thus tend to be small, numerous, and reusable.

Such a world is sufficiently uncommon that it warrants a few introductory remarks:

MEMORY

The execution of a program under Idris is called a "process", a term so fundamental that it is used with a variety of connotations. Abstractly, a process is the elaboration, over time, of the instructions spelled out in a program file. More pragmatically, it is a piece of memory, together with some control registers and secret notes kept by the resident, that is initialized from a program file and that changes as the program executes. Specifically, the secret notes in the resident are sometimes referred to as the process; and at other times, the piece of memory plus control registers that get copied about is called the process.

Regardless, the concept of a memory image evolving over time lies at the heart of the matter. Idris confuses the issue greatly by supporting any number of simultaneously executing processes, so the resident is constantly copying images about memory and even to a "swap" area on disk. The image itself is thus the most concrete thing about a process.

The memory image that a programmer sees comes in two chunks, known as text (instructions, or I-space) and data (variables, or D-space). The text section (chunk) holds all the machine instructions for the program; if produced by a compiler such as C or Pascal, these instructions are further clumped into separate "functions". Idris cares little about this internal structure. It commences program execution by transferring control to the start of the text section, and it reserves the right to disallow changes to the text section (i.e., it may write protect it). That's about it.

The data section is more elaborate. Its lowest (addressed) portion is initialized from the data portion of the program file, parts of which may remain unchanged (constants, tables) and parts of which may be altered in time (variables). A program file may also specify that a certain number of bytes, right above the initialized data area, be set aside and filled with zeros at program startup; this "bss area" merely serves to save space in the program file. Above the bss area is a space reserved for the bss area to be expanded upward, to grow a "heap", and for a "stack" to grow downward.

The stack is a last-in/first-out queue of local storage frames to support recursive function calls. Both C and Pascal make extensive use of the

stack. A heap is a data area from which chunks of storage can be allocated and freed in arbitrary order. Idris supports heap mamagement by maintaining a "data break" address (part of the secret notes kept on behalf of a process memory image); the data break is moved up only upon request of the running program, to make room for more heap storage. Should the heap (data break) and stack (stack pointer) ever meet or cross, the process is deemed out of memory and is aborted. (It is sent the signal SIGSEG, to be precise, which usually forces the process to terminate abruptly.)

There is one further nicety about a process memory image. At program startup, up to 512 characters worth of NUL terminated strings may be written at the very top of memory (bottom of stack), with pointers to these strings pushed on the stack. Such strings are arguments passed to the new program by the program that caused it to startup. They can be ignored or modified as the new program sees fit, but usually they are used as a highly convenient channel for obtaining input parameters.

So this text plus data, together with registers such as the stack pointer and program counter, form the virtual machine upon which a process runs. Depending upon the implementation of Idris, the text portion may be loaded starting at location zero (in I-space), or both text and data may be loaded at separate location zeros (separate I-space and D-space), or neither may truly begin at zero. In the last case, the resident is usually obliged to alter the program image at startup time, to "rerelocate" it for the load address chosen. Once load addresses are determined, however, they cannot change for the life of a program, lest computed addresses be invalidated.

PROCESS

How are processes born? All of them are descendants of an initial "process 1", concocted by the resident at system startup. A descendant is spawned by a program requesting the system to "fork", or make a second process that is almost the identical twin of the requesting process. This alone is not very useful, except that the newer of the twins can request the system to "exec" a new program file, passing it designated argument strings. Here is the genesis of "program startup", mentioned several times above.

So the basic drill is: a running program (process) determines that another program should be run as well. It forks, instructing its child (descendant almost-twin) to exec the new program. The child memory image is overlaid with that of the new program, plus cleared bss area, plus space reservation for heap plus stack, plus arguments passed to it. And the torch is passed on.

There are other kinds of requests that a program can make of the system, about forty of them in fact. The requests are known as "system calls", which can be treated much like (highly sophisticated) machine instructions added to the instruction set of the target machine. Perhaps a third of these deal with process administration; the rest deal with various

aspects of input/output.

The other important system calls in the process administration area are: "exit", to bring a process to a tidy conclusion and report back its status; "wait", to synchronize with a child process performing an exit and collect its status report; "kill", to send a brief but powerful message to a related process; and "signal", to specify how such powerful messages are to be handled. This is pretty much the extent to which processes may directly communicate, except through file I/O. It is usually sufficient.

INPUT/OUTPUT

The Idris Users' Manual has a lot to say about how various forms of I/O can be made to look almost identical by reducing everything to text streams. And the C Programmers' Manual describes a standard system interface which is modelled almost directly by the Idris resident. A few points are worth reemphasizing here, however.

First is the fact that the Idris resident converts between disk blocks and streams of bytes for the programmer, and it optimizes such operations across time and across multiple processes. Per-process memory space thus need not be cluttered with multiple buffers to achieve the joint goals of simple I/O requests and efficient operation.

Then there is the hierarchical filesystem, which makes it easier to impose structure on collections of files and which greatly simplifies the rules for forming filenames. The protection machinery is largely concentrated in the file access modes, so it is easier to make a system secure.

About half the I/O system calls deal with files by name. All use exactly the same conventions for specifying filenames to the resident. The remaining I/O system calls deal with "file descriptors". Like those in the standard C system interface, Idris file descriptors are small non-negative integers which specify which previously opened file is to be used for an operation. Any information on the file status is kept in resident memory, so it is far less susceptible to corruption.

To some extent, the underlying structure of file input/output shows through: directories are conventionally read as ordinary files, in the absence of any special system calls for walking directories; the structure of a filesystem "inode" is made visible as part of the status returned for a file; and internal codes for physical devices appear in file status and in conjunction with special file inodes. Nonetheless, most of this information can be, and is, largely ignored by the bulk of programs that run under Idris.

Some of the generality of the Idris environment is cheerfully sacrificed in the interest of standardizing the requirements for a program. Here are a few arbitrary limitations that are almost universally adopted:

Three file descriptors are assumed to correspond to opened files at program startup: STDIN (0) is the standard source of input, STDOUT (1) is the standard destination for output, and STDERR (2) is the standard destination for error messages. It should be possible to open upwards of a dozen more files, simultaneously within any program.

There is always at least one argument string at program startup. It is taken as the name by which the program was invoked.

Filenames should not be longer than 64 characters, counting the terminating NUL. Text lines should not be longer than 512 characters, counting the terminating newline. And the last character of a text file, if any, should always be a newline (i.e. there should be no partial last line).

NAME

Link - using link and related tools

FUNCTION

Several tools are available for building programs, under Idris and for the Idris environment. These are known as "compilers", for translating high level languages such as Pascal and C, "assemblers", for translating machine specific low level languages, and various tools for dealing with "relocatable object modules". All of them focus on delivering up to a program builder, called link, all the pieces needed to put together a program file that is digestible by the Idris exec system call.

The approach supported by these tools is a traditional one, centered around the concept of "separate compilation", much like the environment implied for FORTRAN or PL/I. It is by no means the only way to build programs. There are interpreters, for languages like Basic and APL, monolithic language systems, for COBOL and standard Pascal, and more ambitious schemes, for the language Ada. Separate compilation is generally more efficient than interpreters, more flexible than monolithic systems, and much simpler than the ambitious approach. Its major drawback is that it imposes a different, and usually much weaker, set of rules for combining modules into a program than for combining functions into a module.

At any rate, separate compilation permits a program to be put together, as described above, from an assortment of pieces. Some of these pieces can be the output of a C compiler, still others can be generated from hand written assembler code. Some modules can be linked unconditionally, still others can be selected only as needed from libraries of useful functions. All input to link, regardless of its source, must be reduced to standard object modules, which are then combined to produce the program file.

The same version of link is used, by the way, regardless of target machine. The standard object module header contains a format byte which specifies gross properties of the target machine, such as byte order within words and word size. link adopts the first format byte it encounters and uses it to adapt to the associated target environment.

One of the great strengths of the Idris link utility is that its output is, or can be, in the same format as its input. Thus, a program can be built up in several stages, possibly with special handling at some of the stages. Such special handling is rarely if ever needed in building a program for execution under Idris, but link is much more ambitious. It can be used, for example, to build programs that can run free standing, such as the system bootstraps and the Idris resident. It can be used to build programs for execution under other operating systems, such as CP/M. And it can be used to build shared libraries, or to make ROM images that are loaded one place in memory but are designed to execute somewhere else.

As a side effect of producing files that can be reprocessed, link retains information in the final program file that can be quite useful. The symbol table, or list of external identifiers, is handy when debugging programs; the programming utility db makes use of this, and the utility rel can be made to produce a readable list of symbols from an object file. There are also "relocation bits", information on how to alter the instruc-

tions and initialized data if they must be moved to a different place in memory. db uses the relocation bits to refine its output when disassembling machine instructions; and some versions of the Idris resident demand such information so they can have more flexibility in utilizing memory. Finally, each object module has in its header useful information such as text and data sizes.

On the other hand, link produces no memory map (the rel utility comes closest to doing that). It has no provision for overlays. And it can only deal with code contributions to two sections, text and data. link should be thought of as simple, but flexible.

IDRIS

Building a program for execution under Idris is straightforward. The text and data sections of Idris correspond exactly to those manipulated by link, and compilers and assemblers are predisposed to put machine instructions in the text section and variables in data. There are even entries in the object file header (oddly enough) for specifying bss and heap plus stack sizes.

The major concern, for programs written in C or Pascal, is that the events at program startup don't mesh well with the high level language environment. Control is simply transferred to the start of the text section, with some pointers to strings pushed on the stack. This must be translated into a call to the C function main, with information on the argument strings as arguments to main. And if main returns, its return value must be used as the status to provide on a call to exit. Such minor adaptation is performed by a small runtime startup sequence, coded in assembler, that is carefully fed first to link, so that it ends up at the start of the text section.

System calls are all packaged as separately compiled modules, each callable from C, which are collected into ordered libraries. Libraries are constructed by the programming utility lib, whose file format is known to link. Moreover, link presumes that any library file it encounters, among its file arguments, contains modules that are to be loaded only if there are pending references to undefined symbols, which references can be satisfied by the library module in question.

What this means is that a large corpus of modules can be assembled, each of which is occasionally useful but all of which, taken together, would be too much of a burden to inflict on every program. The programming utility rel can be used to produce a module-by-module list of symbols defined and symbols required. This list is fed to the programming utility lord, which determines an ordering of the modules such that later ones in the sequence don't require earlier ones. The ordering produced by lord is fed as instructions to lib to construct a library, suitable for conditional loading by link.

Got that?

So the sequence of files presented to link is: header file to adapt to C environment, program specific object files, then one or more libraries of object modules implementing system calls and other generally usable functions.

All of this machinery is well hidden, by the way, by the compiler driver scripts provided with Idris. They see to it that, given simple instructions about the files involved, various source files get compiled to object modules; then they arrange a call to link which brings all the necessary items together to build a program file.

SEE ALSO

The programming utilities are documented in Section II of the C Interface Manual for each target machine.

NAME **Whitesmithing**

Compile - using the multi-pass compiler driver

FUNCTION

Compilers under Idris each consist of three or more separate programs, which must be run in sequence, and which communicate via intermediate files. The multi-pass command driver, c, standardizes the way in which the components of a given compiler are run, and eases the process of changing standard parameters for existing compilers, or extending them, for new ones.

Any driver meant to automate the invocation of a multi-pass program like a compiler should perform at least the following tasks:

1) naming the programs (such as compiler passes) to be run, and running them in the correct sequence.

2) specifying invariant parameters, such as flags, that are to be passed to a given program every time it is run in this particular application.

3) associating user-specified parameters, such as optional flags or source filenames, with at least the first program in the series to be run.

4) passing the output from one program to the next program in sequence, which must accept it as input.

5) end processing, which must include at least the naming of the final output file so that the user (or a later processor) can easily find it, and which might include the running of an optional linker or loader program over all of the files earlier processed.

c performs these tasks by reading an ordinary textfile, or "prototype script", containing a series of prototype command lines, each of which names one of the programs c is to run. Usually, a separate prototype script is provided for each kind of compilation to be done, and each script is given a mnemonic name, with the suffix ".proto". By convention, the script to perform native C compiles for the host machine is called "c.proto"; one to do Pascal cross-compilation for the 8080 might be called "pcx80.proto", and so on. The names can be anything at all; only the ".proto" suffix is normally required, and even that convention can be overridden when c is run.

Given the existence of these scripts, c is then installed by creating one link to it (i.e., an "alias") corresponding to each script, and sharing the same name. When c is run, it examines the alias under which it was invoked, and (by default) tries to find a corresponding prototype script. which it looks for using the same rules by which the shell searches for a command. In the standard case, what happens is this:

1) the user types to the shell one of the aliases under which c is installed (like c or pcx80), followed by the names of the source or object files to be compiled or linked.

2) if the alias given contained a slash (i.e., if it was a pathname), the shell tries to execute c under exactly that name; otherwise, it looks for a file with that name in each directory on the user's execution path.

3) once the specified alias for c is found, c is run, and examines its command line to discover what alias was given.

4) by default, c then appends ".proto" to the given alias, and mimics the shell: if the resulting name contains a slash, then c tries to read a script having exactly that filename; otherwise, it looks for a file with that name in each directory on the execution path, and reads the first one that it finds.

Most of the time, both the aliases for c and the corresponding scripts reside in the same system-wide binary directory (such as /etc/bin). However, c permits two other arrangements, which can be quite powerful if carefully used. Additional links to c may exist in other (perhaps private) directories, with their own scripts; or local scripts may be created with the same name as "official" ones, and privately installed. So long as these private directories are placed on the user's execution path, they will be read automatically whenever c is invoked.

Once the script has been located, c executes in sequence the programs named on successive lines, passing to the first program the name of one of the files that the user originally gave to c, then passing to each subsequent program the name of a temporary file produced by the previous program. This process is repeated for each filename the user specifies.

A line in a script has one of the following formats:

<prefix> : <pname> <pargs> [: <pargs>]

or

<prefix> :: <pname> <pargs> [: <pargs>]

In either format, <prefix> is a string of characters that is matched against the suffix of each command line filename (i.e., the characters in the name following the rightmost dot). Thus a prefix of 'c' would match filenames ending in ".c", 'o' would match names ending in ".o", and so on. Some conventional suffixes are:

Suffix	Type of File
.p	Pascal source
.c	C source
.s	assembler source
.o	object file (assembler output)

However, any non-null string may be used to suffix input files, and be given as a prefix; multi-character strings are perfectly acceptable.

The rest of each prototype line is used to construct a command line that c will execute for each matching input file. <pname> specifies the name (generally the full pathname) of the program to be run. The first group of <pargs> is one or more strings that will be used as the opening arguments of the command line executed. The first string becomes argument zero, the program name actually passed to the program. The colon following, if present, marks the point at which each user-specified filename will be inserted in the command line, while the second group of <pargs> is zero or more strings that will follow the filename on the command line. If the colon and second <pargs> are omitted, then each input filename is appended to the command line. Here, for instance, is a typical script for native C compilation on a PDP-11:

```
c:/etc/bin/pp   pp -x -i /lib/¦../
:/odd/p1    p1
:/odd/p2.11 p2
s:/etc/bin/as.11    as.11
o::/etc/bin/link    link -et_etext -ed_edata -eb_end -lc.11 /lib/Crts.11
```

For each file specified by the user, c searches the script for a line whose prefix matches the file's suffix. It inserts the filename as instructed into the matching line, then starts executing the commands in the script from that point; any previous lines are skipped. The script shown might begin execution at any of three places, depending on the suffix of each input file. For ".c" files, it would start at the first line (which has the matching prefix 'c'), while for ".s" files it would begin at the fourth line, and for ".o" files at the last one.

Thus, if this script were installed at c.proto, and the user typed:

```
% c prog.c
```

the first line of the script would match the input file, and c would create, then execute, the following command line:

```
pp -o /tmp/xxxxxc1 -x -i/lib/¦../ prog.c
```

c adds the flag -o in order to specify the name of a file in which the program being run will place its output (which means that any program used under c must accept the -o convention). If another program is still to be run, then this filename is passed to it as input. Here, if pp returned success, p1 would be executed, with this command line:

```
p1 -o /tmp/xxxxxc2 /tmp/xxxxxc1
```

Then, presuming p1 returned success, its output file would be passed to the following program, and so on. If a program being run returns failure, then the rest of the script is skipped for the current file, and c moves on to the next file given on its command line.

Assembler source files could also be assembled with this script. For each ".s" file given, c would skip the first three prototype lines, and begin by executing the fourth one, thus running as.11, but not any of the compiler passes.

By default, c acknowledges the start of processing of each input file by outputting the filename to STDOUT, followed by a colon and a newline. When (and if) the link line program explained below is run, c outputs its name in the same manner. The flag -v causes c to output each command line generated as well, as it is executed. This verbose option can be used to double-check new scripts, or just to see exactly what command series c generates for a given input file:

```
% c -v prog.c
```

The final line in the script shown is a special one, called a "link" line, as indicated by the double colon following its prefix. The program named on a link line is not run for each command line file. Instead, the output files resulting from each command line file are stored on the link line; then the link line is executed after all the command line files have been processed, presuming that all of the processing succeeded. If any execution of a previous program failed, then the link line program is not run.

The link line accumulates three kinds of files: those output by the preceding program in the script, command line files whose suffix matches its prefix, and any files that are unmatched by the other prototype lines. Hence, the line in the script above would accumulate all the files output by as.11, and ".o" files named by the user, as well as any files with names not ending in ".c" or ".s".

A link line has one other special characteristic, namely that each file output by the program immediately preceding it is made permanent, rather than being temporary like the files output by prior programs. The permanent file output is given the same name as the input file originally specified by the user, but with its suffix changed to be the prefix of the link line.

For instance, the use of c shown above would normally result in as.11 being executed with the command line

```
as.11 -o prog.o /tmp/xxxxxc1
```

The name of the assembler's output file would be taken from the input file prog.c, and its suffix, from the prefix 'o' of the link line.

The link line thus serves as a "terminus" for all previous processing, since it causes the final output from each command line file to be separately stored in a permanent file. Because of this, and because it accumulates all files that "fall through" to it from prior lines in the script, any link line present must be the last line in a script. However, any line in a script having a non-null prefix can be treated as a terminus for a particular run, simply by giving its prefix to c. Then, each time the preceding line is executed, its output will be directed to a permanent file with a suffix given by the prefix of the terminus. c then skips any lines remaining in the script, and starts processing its next input file.

If, for example, the above script were run by typing:

```
% c +s prog.c
```

The line labelled 's' would be considered the terminus of the pass through the script, and so the preceding line would create a permanent output file named prog.s, which would contain the symbolic assembly code generated for prog.c. Neither of the last two script lines would be executed. Similarly, typing:

```
% c +o prog.c
```

would cause execution of the script to stop with the line before the link line, which would create a permanent output file named prog.o containing the object code resulting from prog.c.

Note that c expects to submit each of its input files to at least one prototype command. An input file not matching one of the commands preceding the current terminus will cause an error.

Any prefix can play the additional role of execution terminus. In fact, another type of script is one whose final line contains only a terminating prefix, and thus serves solely to provide a suffix for the output files of the preceding line. Such scripts are useful for compilations in which linking together the individual output files is not desirable. For instance, a script to cross-compile C source files under Idris for assembly and loading under VERSAdos might look like:

```
c:/etc/bin/pp   pp -x -dUTEXT -i /lib/!../
:/odd/p1    p1 -al -n8 -u
:/odd/p2x.68k   p2
s:
```

Here, only one usage of the script is possible: each ".c" file given will be processed by the three compiler passes, and the output from each run of p2 will be directed to a corresponding ".s" (VERSAdos assembler source) file.

In addition to the compile-only and link-line scripts presented so far, scripts can be used whose last line is a full prototype line, but not a link line. When such a script is run, the output from the last program is directed to a temporary file, and so the script produces no permanent output files. For example, a script to do C syntax checking only, with no code generation, might be:

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

Since c was designed to automate compilation, it contains features to aid the orderly generation of output files. The flag -o may be used to name the output file of the link line program specified by a given script; and -p may be used to specify a pathname that will be used to prefix all the permanent output files from a given run. Thus the command

```
% c -o prog prog.c
```

would cause the executable file produced by link to be given the name prog, while

```
% c +o -p /ship/c780/obj/ *.c
```

would compile each of the ".c" files in the current directory into a corresponding ".o" file in the directory /ship/c780/obj, permitting the source directory to be treated as read-only. If both -o and -p are given, the prefix is also added to the link line output filename, if the name doesn't already contain a slash. Hence, the command

```
% c -o prog -p/ship/c780/obj/ prog.c
```

would place the executable output of link in the file /ship/c780/obj/prog (and would create /ship/c780/obj/prog.o along the way).

Finally, the normal lookup procedure for a prototype script can be overridden by naming with a -f flag exactly the script desired for a particular run. A script named in this way need not have the suffix ".proto":

```
% c -f myscript prog.c
```

If the script name given is "-", then a script will be read from STDIN.

The most effective way to learn about c is simply to use it, perhaps with -v specified so that its internal operation can be seen. With the command driver, the flexible use of existing compilers is trivial, while the use of new compilers depends solely on setting up a new prototype script. And because scripts are ordinary textfiles, existing ones can be varied at will to obey local conventions. Most important of all, numerous compilers can be run in exactly the same way, with the same options and conventions. Learning c once is learning it for all of them.

BUGS

c does not currently permit flags to be passed to the programs named in a prototype script. The only way to change the flags with which one of the named programs is run is to use a changed version of the script.

NAME

Debug - using the binary editor db

FUNCTION

db is an interactive editor for binary files that is loosely patterned after the text editor e. It is most useful in working with relocatable or executable object files in standard format, but can also be used to manipulate any direct-access file. db makes available different sets of commands and addressing facilities according to what "mode" it is used in:

1) "absolute" mode, to examine or modify an arbitrary file;

2) "standard" mode, to inspect or patch a relocatable or executable file in standard format; or

3) "debug" mode, to inspect a core image generated by Idris, in conjunction with the executable file whose contents it represents.

This essay presumes some familiarity on the reader's part with the commands and addressing conventions of e; where db provides close parallels to them, neither instance will be discussed at length here. Nor will this essay provide a complete summary of all the commands db makes available (which can be found on the db manual page). Rather, the emphasis will be on how db differs from e, and in particular, how it can be used effectively as a truly portable binary editor and debugger.

Like e, db reads a series of single-character commands from STDIN, and writes any output to STDOUT. Just as commands in e may be preceded by line addresses, db commands may be preceded by byte addresses, whose syntax is a superset of e address syntax. A byte address refers to the start of an item, which may be an integer of length one, two, or four bytes, or a disassembled machine instruction. db supports a large subset of the editing commands provided in e, except that they operate on such items, instead of on lines of text. Given this underlying difference, the two editors in fact have very similar user interfaces. In db, editing generally centers around a "current item" (addressed by a dot '.'), the counterpart of the current line in e. And in general, the addresses preceding a given command designate the range of items on which it will operate.

In absolute mode, db considers its input file to be a series of items, of arbitrary length, which may be updated at will. In standard mode, db honors standard object file format, meaning that it restricts the scope of updates to the file being edited, but makes full use of any available symbol table or relocation information in interpreting addresses. In debug mode, db operates similarly, and also provides a set of commands to access run-time information saved in the core file, such as the machine registers and user stack.

Initialization

The mode in which db operates is established at the start of each run. Normally, the file to be edited is specified on the command line, perhaps preceded by flags detailing how it is to be accessed. If the flag -a is

given, or if the command line file cannot be interpreted as an object file in standard format, db comes up in absolute mode, and outputs a heading line noting the fact.

Otherwise, if -c is given, db operates in debug mode, and expects the command line file to be a standard object file; the associated core file must reside in the current directory, under the name "core". In debug mode, db usually accesses only one of the two input files at a time; initially, db accesses the core file.

If -c is not given and the command line file is in standard format, db operates in standard mode (and so edits an object file alone). In debug mode or standard mode, db outputs a header giving the length (in decimal) of the text, data and bss segments of the object file being edited.

For instance, either of the following exchanges would open a disk image directly for editing:

```
% db /dev/rm0
/dev/rm0: absolute
```

or:

```
% db -a /dev/rm0
/dev/rm0: absolute
```

While the following would edit the executable file myprog, and the core file it generated when run:

```
% db -c myprog
myprog: 10438T + 1124D + 0B
```

External Interface

Unlike e, db does not make an internal copy of the file(s) being edited. Instead, it interacts directly with the permanent version of each file, meaning that any changes are made immediately to that version. This also means that db can edit files of arbitrarily large size. In particular, files edited in absolute mode have no pre-defined maximum size; an attempt to access a given item succeeds if the necessary I/O does.

With this significant difference, db supports a set of absolute mode commands for interacting with external files that is analogous to the set e provides. The command 'e', followed by a filename, closes the file currently open for editing and opens the file named. The new file will always be accessed in the same mode as the previous one. The command 'r', followed by a filename, overwrites some part of the current file with the contents of the file named. The command may be preceded with an address specifying the first item to be overwritten; if no address is given, the external file overwrites the current file starting at the current item.

The command 'w', followed by a filename, writes some part of the current file to the file named, which the command always newly creates. The command may be preceded by one or two addresses specifying the range of bytes

that will be output; by default, all of the current file is output. The file named in the 'r' or 'w' commands may not be the file being edited. Finally, the command 'f' may be used to display the name of the current file.

Of these commands, only 'f' may be used in standard mode or debug mode, except that db always permits one initial 'e' command to be given.

Addressing

In absolute mode, the byte addresses used to access the input file closely correspond to the line addresses used in e. Only two differences exist: bytes are numbered starting at zero, and there is no pre-defined "last byte", so that the symbol '$' has only part of its meaning in e. In general, \&'$' can be used only as the second of two addresses specifying a range of items, and can never appear alone.

Byte addresses themselves (as their name implies) designate nothing more than byte offsets within the file. Depending on the current input/output format, the item itself may be of varying length, but the address itself is always counted in bytes.

In debug mode or standard mode, addressing is more complicated. Interacting with an object or core file means examining particular locations within the text or data segment the file contains, and not merely looking at a physical location inside the file. So in these cases, an address has two components: first, an indicator of which segment (text or data) it refers to, and second, the runtime location in "memory" that a given item would start at, if the object file were actually loaded.

db obtains the biases of the text and data segments from the standard file header, whence it also gets their lengths; any bss segment is presumed to immediately follow the data segment. Two ranges of addresses are then accessible: those in the text segment, which extend from the text bias up for as many bytes as the segment is long, and those in the data/bss segment, which extend similarly upward from the data bias. An access to a given address is mapped to the physical offset in the object or core file to which the address corresponds; the segment component of an address unambiguously indicates to which range it belongs.

In an object file, reading an address in the bss region returns all zeroes, and addresses in that region cannot be written to. In a core file, the data/bss segment is interpreted differently. The data segment extends from the core file data bias up to the "system break" (i.e., the highest dynamically-allocated storage) in use at the time of the dump. The bss segment extends from the system break up to the end of the user process image (i.e., the base of the runtime stack, which extends downward from the "top" of memory).

All residents currently allocate to a process a single swatch of memory, so that its data segment immediately follows its text segment. No overlap or gap between the two ranges of addresses is possible. Object and core files under an 'R' resident will always have a text bias of zero, since exec-time relocation is unnecessary. Under a 'B' or 'S' resident,

however, core files will always have a non-zero text bias, which is the actual location at which the process ran, since the resident makes no use of memory management hardware. And object files may be linked with a non-zero text bias for reasons of efficiency.

When the current output format is machine instructions, these complications are largely hidden from the user. If a symbol table and relocation streams are available in the object file, the disassembler will always output relocatable addresses as a symbol name plus a decimal byte offset, so long as a symbol exists with a value less than or equal to the address, and the same relocation base. If no relocation information is present, any symbol table available will still be used, but constants imbedded in the instruction stream may be mistaken for addresses, since db will have no way to tell them apart.

In debug mode, db presumes that the text and static data areas of a core file are "parallel" to those in the original object file, so it is able to find relocatable addresses in the core file by using the object's relocation streams. If the core file has a different text bias than the object file, the offset between them will be applied as needed to make the object file symbol table usable for core file address references. db will not tell an out and out lie -- if an address from such a core file cannot be output symbolically, then db will output its numeric value as stored in the file. But if the address, when altered, can be output symbolically, then the symbolic form will be used.

Address Terms

Addresses in db are composed of one or more address terms, which may be combined by the same rules e uses. However, db provides a slightly expanded set of terms. First, in standard or debug mode, \&'$' always refers to the last address of the current segment, text or data/bss. Since in this context '$' has a well-defined value, it can be used by itself, though items cannot be read past the end of a segment.

Also, when a symbol table is available in the object file being edited, db permits symbols to be specified as address terms. A symbol specifies the address location indicated by its value, and has the segment component indicated by its relocation base. A text-relative symbol refers to the text segment; data or bss-relative symbols refer to the data/bss segment.

Not all address terms have explicit segment components. An offset from '.' or '$' refers to the same segment as dot currently does. A symbol that is not text, data, or bss-relative, or a number appearing alone as an address, refers to the same segment as the last previous address given for the same command, or to the segment that dot refers to, if no previous address was given.

In both standard and debug mode, db provides three constants to simplify the use of object or core file addresses. The constant 'T' has the value of the first address in the current text segment, while 'D' and 'B' have the value of the start of the data segment and bss segments, respectively. All three also have the appropriate segment component. Two main uses for these constants are to ease access to an item at a given offset from the

start of a segment, or to force the current item address (dot) into one segment or the other, so a subsequent command will work properly. Thus to look at an item 10 bytes into data segment, you might type:

```
D10 l
```

while to set dot to the start of the text segment, and then print several instructions, you could use:

```
T:+50 pm
```

Finally, a note about numbers: db outputs all numbers that are part of an address using the conventions of the language C. That is, numbers in hexadecimal are preceded by "0x", numbers in octal by "0", and numbers in decimal by nothing at all. The same conventions are used for numbers input as address terms. However, these conventions apply only to addresses; the contents of an item are output without a prefix, even when output in hexadecimal or octal. Likewise, items must be input without C-style prefixes. (See Editing, below.)

Items

Items are the units into which db divides the data in its input file, which means they're also the units on which many editing commands operate. However, an "item" is not defined statically, but in terms of the current input/output format, which can be changed at will. This format specifies the length of an item (one, two, or four bytes, or the word length of the target machine), and defines how each item is to be converted on input or output. Items can be treated as integers in the three common bases, or as strings of ASCII characters. Alternatively, items may consist of disassembled machine instructions, output in conventional symbolic form. An input/output format is named as a base code followed by a size code:

Base code	Item output as:	Size code	Item length:
a	ASCII characters	c	char (1 byte)
h	hexadecimal	s	short (2 bytes)
o	octal	i	int (2 or 4 bytes)
u	unsigned decimal	l	long (4 bytes)
m	machine instruction		

A base code may be given with no size, in which case the size defaults to int; or a size code may be given with no base, causing the base to be taken as signed decimal. The base code 'm' may not be followed by a size, since the size of each item in this format is just the length of the disassembled instruction. Thus a format of "al" defines long items output as strings of ASCII characters, while a format of 'h' calls for int items to be output as hexadecimal integers, and 's' calls for short items output in signed decimal. Note that the base code 'a' converts characters like the 'l' command in e. Characters with values in the range [0, 7] are output as "\[0-7]", those in the range [8, 13] are output as "\[btnvlr]", and any other character that is not printable ASCII is output as a '\' followed by the three-digit octal number giving its value.

Editing

Items can be examined with the db commands 'l' and 'p', which are rough analogues to their namesakes in e. In db, the difference between the two is that 'p' (for "print") sets the current item address (dot) equal to the last location examined, while 'l' (for "look") leaves dot unchanged. The current input/output format may be changed by following either command letter with the appropriate base and size codes. For instance, to print three shorts in hexadecimal, starting at the current item address:

```
.,+5 phs
0x100   81cc
0x102   0002
0x104   1401
```

When, as here, items are treated as integers, db converts as necessary from the native byte order of the target machine. When items are being output in ASCII, however, each item is treated as a array of characters, which are output in order of increasing index in the array. Thus, depending on the target machine, the order of the characters output may differ from the order in which the same bytes would appear in an integer output format. Here the results of four examinations of a long written for a PDP-11 (using db11):

```
. lal
0x200   abcd

. lhl
0x200   62616463

.,+3 lhs
0x200   6261
0x202   6463

.,+3 lhc
0x200   61
0x201   62
0x202   63
0x203   64
```

Note that the '.' preceding the first two 'l' commands is unnecessary, since by default 'p' or 'l' display only the current item.

Items may be changed with the commands 's' and 'u'. The 'u' command (for "update") is the single db equivalent of the e commands 'a', 'c', and 'i'. It replaces some series of items in the file being edited with items read from STDIN using the current input/output format. Items are read as ordinary text strings, one per line, until a '.' is read on a line by itself. Each line may contain only characters legal in the current format; each is converted to exactly one item of the correct length. Again, prefixes are not recognized here, so items entered in hexadecimal should not be preceded with "0x" (nor octal ones with a "0").

db will convert integers to native byte order before writing them to the file. Integer values too large to fit in the current item length will be truncated without comment. If the input format is ASCII, however, a line specifying too many characters to fit in an item will not be accepted. Also, if input is given in this format, a line containing too few characters to fill an item will be right padded with NULs.

The 's' command closely resembles its counterpart in e. It is followed by a target string to be searched for, and a second string that will replace any occurrences of the target. As in e, the target may be a regular expression. The command then converts each item it examines to output representation (using the current input/output format), replaces any occurrences of the text of the target string with the text of the replacement string, and converts the result back to internal format. In that form, it replaces the original item, by the same rules as given above for the 'u' command. Note that in db the 's' command always outputs the revised item that contained the last instance of the target string. (That is, in terms of the 's' command in e, the command in db always acts as if it were suffixed with 'p'.)

Items cannot be updated as machine instructions, since the disassembler has no counterpart for interpreting input assembly code. Hence, neither 's' nor 'u' may be used if the current input/output format is machine instructions.

NAME

Headers - standard include files

SYNOPSIS

```
#include <std.h>
#include <sys.h>
   etc.
```

FUNCTION

Information to be shared among C source files is conventionally concentrated in one or more "header" files, to be #include'd as needed at the top of each source file. All such files provided with Idris are in the standard library /lib, and the various C compiler drivers are wired to search /lib for any include files written in angle brackets.

Of all these files, std.h is by far the most important. It is included in all source files for the compilers and Idris. It defines the notation used throughout these manuals. And it provides definitions used by all the other header files. (See Section II of the C Programmers' Manual for a description of its contents.) Consequently, it should be included in every source file written to interact with the standard C environment, and it should be named before any other files.

The file sys.h plays a similar role, for programs that must interact with the Idris resident. It defines all the constants and data structures that cross the interface between programs and the resident. And it provides definitions used by the more specialized header files. (See Section II of this manual for a description of its contents.) Thus, it should be included in every source file written to interact with the Idris environment, and it should be named right after std.h.

The complete list of header files provided with Idris is:

bio.h block I/O within the resident. Used by some device handlers.

cio.h character I/O, primarily tty style, within the resident. Used by some device handlers.

cpu.h process management, within the resident. Used to interpret files such as /dev/myps and /dev/ps.

dump.h dump file format. Used by dump and restor.

fio.h file I/O, within the resident. Used to interpret files such as /dev/inodes and /dev/mount.

ino.h filesystem format. Used by programs that must manipulate filesystems directly.

mount.h mount file format. Used by programs that must manipulate the external mount table.

pan*.h panel layout for registers within a core file, dumped by the resident. Used by the resident and db to interpret core dumps. There is

a different file for each target machine, such as pan11.h, pan80.h, etc.

pascal.h file I/O, within the Pascal runtime. Used by Pascal runtime library to administer files.

pat.h pattern matching codes. Used by programs that use the standard library functions amatch, makpat, and match to match regular expressions.

res.h resident conventions. Used in all resident source files to define system wide constants and types. Sort of an internal sys.h.

sh.h shell conventions. Used in all shell source files, and by programs that invoke other programs via the standard Idris functions xecl and xecv.

sort.h sort key conventions. Used by programs that use the standard library function getkeys to define sort keys.

std.h the universal header file, as described above. Used to implement the standard C environment.

sup.h filesystem format. Used by resident source files that must manipulate the filesystem directly. Sort of an internal ino.h.

swp.h timer and space management structures, within the resident. Used by various Idris schedulers.

sys.h the Idris interface header file, as described above. Used to implement the standard Idris environment.

time.h time and date structure. Used by the Idris support library functions atime, ltime, and vtime to manipulate time information.

usr.h per-process information swapped with the process image, within the resident. Used by resident source files that must manipulate the process image.

who.h log and who file formats. Used by programs that must manipulate the administrative files /adm/log and /adm/who.

Most of these files can be ignored most of the time. But when a program must interact with an existing corpus of code, including the relevant header files is a safe and convenient way to enforce the necessary conventions.