Many facets of the system are configurable. Look inside the file `system.setup` for some clues.

If you are using this on rabbit, there are two ways to run the emulator. The most direct is just to type the command "`run` filename", where filename is replaced by the name of your executable .exe or .dll file. You do not need to type the .exe. This will just run your program in the way everyone would expect. If everything goes well and your program reaches its end, the emulator also exits and you are back to the Unix command prompt. If something does go wrong, the emulator will switch to single step debugging mode. It shows you the current state of the computer, gives you a > prompt, and allows you to enter commands as described below. If it doesn't immediately show you the current state of the computer (register and flag values and next instruction) just press the enter key.

The other way to start the emulator is with the command "`computer` filename". This just gets everything ready to run, but does not start your program. It just gives you the > prompt from above, and accepts your commands. This is useful for debugging: you can set breakpoints and various other things, and start the program running whenever you are ready. Or you can just single-step through your program from the beginning. Single stepping involves just executing a single machine code instruction at a time. The emulator is unaware of how these instructions relate to your BCPL program.

If you wish any .exe files to be automatically loaded into memory at startup, there are two options. Either add a line like this "`load file.exe at 1400`" (1400 is the address where the file would be loaded, in hexadecimal) or "`load file.exe`" to `system.setup`, or run your program with the `run` command above. By default the first file will be loaded into memory starting at address 0x400 (1024). Subsequently loaded files will have a default loading address immediately after the end of the previously loaded file.

Inside the emulator, nearly all numbers are assumed to be hexadecimal, there are a few exceptions and they will be pointed out. If you wish to change the base, just precede the number with `0d` for decimal, `0o` for octal, `0b` for binary, or `0x` for hexadecimal. If the number is negative, the - sign must be before any such prefix. When the emulator displays a number in hexadecimal, it uses the lower case letter 'o' instead of the digit zero. This makes large addresses less cluttered to the eye. When you enter numbers as parts of commands, it accepts the letters 'o' and 'O' as a zero.

Control-C is properly trapped. If a program is running, control-C instantly stops it and puts the system in single step debugging mode. If no program is running, control-C is harmless. BUT two consecutive control-Cs will exit the system. This is just in case I screwed up the control-C trapping. We don't want any unstoppable programs. This means that when a program is running it will take three consecutive control-Cs to completely exit everything.

when the emulator is accepting commands, with the > or `stepping>` prompts, the keyboard arrow and backspace keys have the effect you will be familiar with from modern unix shells for repeating or editing previous commands. control-A moves the cursor to the beginning of the line, control-E to the end. control-D deletes the character to the right of the cursor, and control-X discards the entire command being typed.

While a program is running (not in single step mode, but properly running), everything typed at the keyboard is instantly captured, without waiting for enter to be pressed. If interrupts are enabled (i.e. the `$INT` flag is on), the `KEYBD` interrupt is caused.

There are a few flags that can be specified with the `run` or `computer` commands:

`-q` (quiet) prevents initial setup information from being printed at startup. The "`run`" command is just an alias for "`computer -rq`".

`-r` makes your program run automatically, and also also turns on auto-exit, which means that if the program reaches a natural end, the emulator exits too.

`-s` turns auto-exit back off.

`-z` normally the emulator uses the letter '`o`' instead of the digit `0` when printing hexadecimal numbers. The `-z` option makes it go back to using real zeros instead.

Any combination of `-q`, `-r`, `-s`, and `-z` may be joined together, e.g. `-qs`

`-c` provides command-line input to the program, like "`run prog -c string`" or "`run prog -c "one two three"`", the strings are provided to the running program as a parameter: you can declare start with a parameter, and it will be set to point to a nil-terminated array of strings.

`-i filename` means that all input normally taken from the keyboard will be read from the named file instead.

`-p filename` starts a script. Initially commands will be taken from the named file instead of the keyboard, when the end of the script file is reached the system reverts to taking commands from the keyboard.

`-ti` means that all interrupts will be traced. While a program is running normally or single stepping, every interrupt that is received and every `IRET` (return from interrupt) instruction that is executed will be reported, along with information on anything that goes wrong with interrupt processing.

`-tc` means that all function calls and returns will be traced.

`-ign address` exempts one single function from being traced if `-tc` has been specified. If a call is made to the given address, it will not be reported, and neither will the corresponding return. `-ign` may be used at most four times, so no more than 4 call addresses can be exempted.

Many of the commands require some information to do their jobs. This will usually be a number or an address. Addresses are usually just numbers, default hexadecimal. Some commands also accept "operands". Operands can be based on the values in registers or memory locations, or the addresses of functions or files. In the following description, certain symbols will be used to represent a variety of different possible inputs:

*R* stands for any register or special register, such as `R4`, `PC`, or `TIMER`.

*FL* stands for the letter `F` immediately followed by a flag name, e.g. `FVM`.

*NH* stands for a number, default hexadecimal.

*ND* stands for a number, default decimal.
(the defaults for bases can be overridden by beginning the numbers with `0x`, `0d`, `0o`, or `0b`.)

*SYM* stands for the name of a function or global variable from your BCPL program.

*FILE* stands for the name (without extension) of one of the BCPL files or imported libraries or assembly language files that your program is made from.

*EXE* stands for the name of any `.exe` or `.dll` file that has been loaded into the emulator. These *EXE*-type files may be built from a large number of *FILE*-type files. The `SYM`, `FILE`, and `EXE` forms only work if the files were compiled and linked without the `-d` command line option, see the description of the `DEBUG` command below for details.

An operand can be any of:

| | |
|---|---|
| *NH* | which represents a number is hexadecimal |
| *ND* | which represents a number is decimal |
| *R* | (R1, R2, SP, etc) which represents the value of that register, |
| *R*+*ND* | which represents the value of the register plus the number, |
| *R*−*ND* | which represents the value of the register minus the number, |
| !*NH* | which represents the contents of memory location *NH*, |
| !*R* | for the contents of the memory location whose address is stored in the register |
| !*R*+*ND* | for the contents of the memory location whose address is *ND* more than the value of the register, |
| !*R*−*ND* | for the obvious counterpart to !*R*+*ND*, |
| *FNAME* | which represents the value of the flag named *NAME*. |
| `LOCAL`*ND* | for the address of the current function's *ND*th local variable, e.g. `LOCAL7`. |
| `PARAM`*ND* | for the address of the current function's *ND*th parameter, |
| !`LOCAL`*ND* | for the value stored at `LOCAL`*ND*, i.e. that variable's value, |
| !`PARAM`*ND* | for the value stored at `PARAM`*ND*, |
| ?*SYM* | for the address of that variable or the address of the function *SYM*'s first instruction, |
| ?*FILE*/*SYM* | the same, but used to disambiguate when a function of the given name appears in more than one source file, |
| ?*EXE*/*FILE*/*SYM* | to disambiguate further when two source files of the same name were included in more than one loaded executable. Such as functions from the "io" library, which is imported by nearly everything. |
| `F?`*FILE* | for the address of the first word occupied by code from that source file |
| `F?`*EXE*/*FILE* | to disambiguate as before. |

All of these forms, as all command names, are case in-sensitive. Every BCPL program has a hidden function that is called before `start`. It is sometimes needed to initialise some global variables. This function is referred to with the name `(fixes)`.

There are three modes of behaviour: running (the program just runs and gets on with things, so long as nothing goes wrong. Your only interactions are to provide normal input if it is requested or to type control-C to change to single step mode), command (the

program is not running, but can whenever you want. You just type commands to the emulator), and single stepping (the program is sort of running but frozen. Each time you press enter, a single instruction is executed. You can type any command you like. The current state of the CPU is always displayed and most effects of instructions that are executed are shown). If the prompt you see is just >, you are in command mode and pressing enter will change to single stepping mode. If the prompt is `stepping>`, you are in single stepping mode. Pressing enter is the same as typing the command `STEP`, the command `x` returns to command mode. All commands are case insensitive.

General:

`R` or `RUN`:

Continue (or start) running the program without debugging. Just run normally as though the "`-r`" command line option had been used. In other words, behave like a real computer running normally.

`R` *ND* or `RUN` *ND*:

The same as just "`R`" or "`RUN`" above, but after the given number of instructions have been executed, the system returns to single step mode and waits for another command.

`S` or `STEP`:

Switch to single stepping mode if not already there, and execute one instruction

`S` *ND* or `STEP` *ND*:

The same as just "`S`" or "`STEP`" above, except that it continues to execute instructions, showing the state of the CPU before each one, until the given number of instructions hve been executed.

`START`:

When single stepping through a program produced by the BCPL compiler, this should be used once before `STEP` is used. It runs silently through the initialisation code for static variables, and returns to single step mode when the first “real” instruction of the program, the call to `start()`, is reached. Of course, if you want to debug the initialisation code itself, just don't type `START`.

`ENTER` (meaning just pressing the enter key):

Just press Enter and the system goes into single step mode. It shows the values of all the registers, and the next instruction to be executed, and lets you take control.

Registers 0 to 12 are printed in decimal if they are four or fewer digits long, and the instruction's numeric operand part is printed in decimal too, everything else is in hexadecimal.

If you are already in single step mode, pressing enter will execute just one instruction then wait for another command. If you type the command "`x`" it will leave single step mode and wait for another command. All other commands are equally available in or out of single step mode.

`X`:

In single step mode, changes to command mode. In command mode, exits from the emulator.

`Q` or `QUIT` or `EXIT`:

Exit from the emulator.

`OBEY filename`:

Start reading instructions from the given file. Every line in the file will be treated exactly as though you had typed it in response to the emulator's prompt. When the end of the file is reached, the system reverts to taking commands from your keyboard. Scripts may not be nested: no script file may contain an `OBEY` command.

`ECHO anything`:

The same as the Unix shell command, it just prints whatever follows. Useful inside scripts.

`SH`:

Starts a new Unix shell so that you can do Unix things without having to exit from the emulator and lose everything you are doing. Exitting from the shell in the normal way brings you back to the emulator.

Basic examination and manipulation of the state of the computer

`ALL`:

The values of all the special registers are shown. All are displayed in hexadecimal because they represent memory addresses, except for `EXITCODE` and `CGLEN` which are shown in decimal.

`DEC`:
`HEX`:
`FLOAT`:

In single stepping mode when the state of the CPU is shown before each instruction is executed, the frame pointer, stack pointer and program counter are always shown in hexadecimal. The other registers are shown in decimal if they have small values and hexadecimal if they are large (large numbers are more likely to be addresses). These three commands make the state be displayed again, but this time with all values shown in decimal, hexadecimal, or interpreted as floating point values respectively.

`SHOW operand` or
`? operand` or
`operand` alone:

The value of the operand is reported. Generally, the value of the operand itself, along with the contents of the memory location that it points to, are both reported. If the operand is based is based on the PC register, then the memory contents are also displayed as decoded instructions. The `?` and `SHOW` forms are provided for

the rare cases when an operand alone could appear to be another command, or the beginning of one, particularly when the operand itself begins with a `?`.

`SHOW operand TO operand` or (can use − instead of `TO`)
`? operand TO operand` or
`operand TO operand`:

Show is applied to the whole range of memory locations given. If the first operand is greater than the second, then are shown in reverse order. With this usage, everything is treated as a memory address: `SHOW R2 TO R7` will not show the values of registers 2 to 7, it shows the contents of memory between the locations stored in `R2` and `R7`. `TO` may be replaced by a dash `-`, but it must still be surrounded by spaces.

If any of the addresses shown is the same as the value of `FP`, `SP`, `PC`, or the register that the address range was based on (such as R4 in `SHOW R4-5 TO R4+10`), that is pointed out.

`SHOW operand FOR number` or (can use `:` instead of `FOR`)
`? operand FOR number` or
`operand FOR number`:

The contents of the number (default decimal) memory locations starting from the address given by the operand are displayed. If the number is negative, it starts at the given address and works backwards by that number of memory locations. `FOR` may be replaced by a colon `:`, in which case spaces are not required.

`DECODE address` or
`DECODE address TO address` or (can use − instead of `TO`)
`DECODE address FOR number`: (can use `:` instead of `FOR`)

Exactly the same as SHOW, except that everything shown is always also shown decoded as an assembly language instruction.

`= expression` or
`expression` alone:

The value of the expression is calculated and displayed in hexadecimal and decimal. The expression is in "reverse polish" format. It consists of values and operators, and we start processing with an empty stack. The expression is processed strictly left-to-right, no operator priorities or parentheses are needed. Whenever a value is encountered, it is pushed onto the stack. Whenever an operator is encountered, the appropriate number of operand values are popped from the stack, the operator is applied to them, and the result is pushed back. For non-commutative dyadic operators, the left operand should be pushed first.

When the end of the expression is reached, there must be exactly one value on the stack, and that is the result.

The operators are `+`, `-`, `*`, `/`, `rem`, `<<`, `>>`, `!`. The first four do what you would expect, `rem` is the modulo operator, and `<<` and `>>` are left and right arithmetic binary shifts. `!` corresponds to the BCPL operator: it only has one operand, the operand is popped from the stack and treated as a memory address, the value stored at that address is pushed onto the stack.

The values entered may be any operands. A few examples:

| | |
|---|---|
| `= 0d12 0d12 *` | the result is 144 (decimal). |
| `= intvec r3 2 * +` | the result is the value of intvec + twice R2. |
| `= intvec r3 2 * + !` | the result is the contents of that memory location. |
| `= 400 ! 520 ! +` | whatever is stored at 0x400 + what's stored at 0x520. |
| `= FP 3 + !` | is the same as `? FP+3`. |
| `= local1 param3 *` | the first local variable is multiplied by the third parameter. |
| `= PC F?file -` | how many words inside `file.exe` are we now? |
| `= ?g ?f -` | how many words apart are the functions f and g? |
| `= ?f 677 -` | where is address 0x677 relative to the function f?. |

`SET operand = expression` or
`operand = expression`:

This is the counterpart of `?` or `SHOW`, instead of showing what is stored somewhere, it lets you change what is stored there. The = sign in the command is optional. The `expression` is any expression as defined for the = command above. The `SET` form is provided for the rare cases when the first operand could be mistaken for another command. A few examples:

| | |
|---|---|
| `set local2 = 0d66` | the second local variable is set to 66. |
| `set SP = 407` | the stack pointer is set to 0x407. |
| `set 1001 = 0b1010` | memory location 0x1001 is set to ten. |

assuming that register 4 contains 0x6543:

| | |
|---|---|
| `set R4+16 = 407` | memory location 0x6553 is set to 0x407. |
| `set R4+0 = 407` | memory location 0x6543 is set to 0x407. |
| `set R4 = 407` | register 4 is set to 66. |

Debugging: break points, watches, tracing, skipping function calls, etc.

`TRAIL`:

Displays the last 16 values of the program counter and the instructions that are at those addresses. That is “how did I get here?”

`BREAK n address`:

`n` must be `1`, `2`, or `3`. `address` is any operand. Breakpoint `n` is set to the given address. Whenever execution reaches that address, the program will automatically stop running and return to single-step debugging mode. The `R` or `RUN` command can be given at any time to resume execution. A breakpoint is a lot like reaching a `BREAK` instruction (generated by the BCPL builtin `$debug(...)`) in a running program, but uses a totally different mechanism. A program can have as many `BREAK` instructions as you want, and they have no effect on efficiency. But breakpoints set by this command have a small cost in run time, and there can only be three of them.

`BREAK n NEXT`:

Breakpoint `n` is set to the very next instruction in the program. This is only useful when the current instruction is a `CALL` or `SYSCALL` to a function that you do not want to investigate. `BREAK 1 NEXT` followed by `R` will run right through the

function and any others that it calls, but go back to single-step debugging mode as soon as the function exits and control returns to the next instruction.

`BREAK n 0`:

Breakpoint `N` is disabled.

`BREAK 0`:

All breakpoints are disabled.

`OVER`:

Exactly the same as `BREAK 3 NEXT` followed by `RUN`, except that when the following instruction is returned to, breakpoint 3 is set back to its original value. Used to jump over an uninteresting function call when debugging.

`BT` or `BT` *ND*:

Backtrace. The program counter, frame pointer, and stack pointer for the currently executing function are displayed, followed by all of the saved values for those same registers for every function call on the stack. If `num` is provided, that is the limit to the number of saved addresses that will be shown. If `num` is not provided, the limit is 25. This command relies on the assumption that the stack is organised as it is for BCPL programs.

`FRAME` *ND*:

Shows the entire contents of the NDth stack frame. 0 means the information for the current function call, 1 means the one that called it, 2 means the one that called 1, and so on. Everything from the last parameter, through the return address and saved frame pointer, to the last local variable is shown. It assumes that the stack is organized as it is for BCPL programs. There is a limit of 500 words to the size of stack frame that will be shown.

`SETWA address`:

The value of the special register `WATCH` is set to the address, default hexadecimal. As soon as any instruction is about to change the value stored in that memory location, an interrupt is caused. If your program has not enabled interrupt processing, then the program will be stopped and return to single-step debugging mode. It can be resumed at any time with the `R` or `RUN` command. `address` may be any operand, but is always treated as a memory address: `SETWA R4` does not look for changes to the value of `R4`, but changes to the value stored at the memory location pointed to be `R4`. Even if `R4` changes, the watched address will not. For example, `SETWA LOCAL3` will inform you whenever the current function's third local variable's value changes. Watches are computationally expensive, so only one address can be watched at a time. `MONITOR` is similar and can monitor as many things as you want.

`SETWA 0`:

Clear the watch

`SETWA`:

Report on what is being watched

`MONITOR things`:

`things` is a space-separated list of monitorable objects. The things may be names of registers such as `R7` and `FP`, the names of special registers such as `TIMER`, memory addresses (default hexadecimal), or any other operand. If the thing is an operand, it will be interpreted as a memory address. When a thing is being monitored, any change to its value is reported. If you are single-stepping, the report is immediate, but if the program is running normally, the changes will not be reported until it stops for some reason. Anything else would slow things down too much. When a monitor is started, the thing's current value is stored. Whenever a change is detected from the original value, the change is reported, and then it continues to monitor for changes from that new value. If you re-issue a `MONITOR` command for something that is already being monitored, that thing's stored value will be replaced by its current value. To stop monitoring a particular thing, just put a − before its name, for example `MONITOR -TIMER -1400`.

`MONITOR NONE`:

All monitored things are cancelled.

`MONITOR`:

Nothing else on the command line: all currently monitored things are listed along with their stored values.

`TRACE CALLS`

Tracing of function calls and returns is enabled exactly as though the `-tc` command line option had been given. Every time a function is called or returned from, the event is reported. `TRACE -CALLS` turns that tracing off.

`TRACE INTS`

Tracing of interrupts is enabled exactly as though the `-ti` command line option had been given. `TRACE -INTS` turns that tracing off.

`TRACE IGNORE address`

The function that starts at the given address (default hexadecimal) is exempted from function tracing. Any calls to it or returns from it will not be reported. `TRACE IGNORE -address` will remove the given function from the ignored list. `address` may also be any operand, as in `TRACE IGNORE ?OUT`. Only four functions may be ignored at any time.

Small details

`WHAT thing`:

If `thing` is a symbol known to the system, such as a register name, special register name, flag, instruction operation code, it tells you what kind of thing it is, and any value associated with it.

`FLOAT` *NH*:

Shows the floating point value that the given number would represent if it were interpreted in that way

`FIX f`:

The opposite. `f` must be a floating point number. The int that represents it inside the computer is displayed.

`MODE items`:

Items may consist of spaces and the symbols `+`, `-`, `x`, and `z`. Autoexit mode means that when the program reaches its end, the emulator exits too. It is set when the `-r` flag is given to the `run` or `computer` command, and turned back off by the `-s` flag. Zero mode is off by default. Zero mode being on means that when a number is shown in hexadecimal, real zero "0" characters are used for the digit zero. When off it means that zeros are displayed as lower case letter "o"s. This makes reading large values a little easier. The `MODE` command changes those modes. `MODE xz` or `MODE +xz` turns them both on, `MODE -xz` turns them both off, `MODE +x-z` turns autoexit mode on and zero mode off, etc. `+` or space means on, `-` means off. Without any items the `MODE` command reports the current settings.

`INPUT whatever`:

This behaves as though the keys corresponding to the rest of the command line, `whatever`, had been typed to a normally running program: the characters are added to the hardware keyboard buffer. If you press Enter at the end of the command, then the '`\n`' character is added to the keyboard input buffer. If, on the other hand, you end the line by pressing ESC twice consecutively instead of enter, nothing extra is added. There is no way to put an escape character into the input buffer. This command does not automatically generate an interrupt. The point of the `INPUT` command is that when a program is being single stepped there is no other way to provide keyboard interactive input.

`KEYS`:

Every key you press will result in its ASCII code being printed in decimal inside square brackets. This is useful for discovering the codes corresponding to any invisible characters you might want to recognise, or for learning the sequences of multiple characters that arrive when you press an arrow or other mysterious key. Control-c escapes from the `KEYS` command.

`COUNT`:

The number of instructions executed and the total run time is reported.

## Loading executable files and finding their contents

These features let you find variables and functions in the program being debugged. To make the best use of them, you may want to generate an assembly listing when your program is compiled. Giving the `-la` flag to the `prep` or `assemble` commands does this. After `prep -la xyz`, the file called `xyz.las` will contain the assembly listing. Every single word produced is shown in hexadecimal under the assembly language instruction or command that produced it, along with its position in the file. The position is shown as four hexadecimal digits and is relative to the beginning of the file. In simple cases files are usually loaded into memory starting at address `0x400`, so an assembly listing position of `02A5` would correspond to memory location `0x00006A5` in the computer.

`LOAD filename` or
`LOAD filename AT address`

The named file, which is expected to be a properly constructed `.exe` or `.dll` file, is loaded into memory at the given address, default hexadecimal. If no address is given, the file is loaded immediately after the last file that was loaded. If no file has yet been loaded, the address `0x400` will be taken as the default. `address` may be any operand.

`LOADED` or
`LOADED name begin TO end` or
`LOADED -name`:

The first form reports the address ranges (first and last) of all `.exe` and `.dll` files that have been loaded either because they appeared on the command line or because an explicit `LOAD` command was given. Those are the only two ways the emulator can know about a loaded file.

If your own program has used software to load a file itself, you can only know the addresses used if you make your program tell you them. But then you can use the second form of this command to make the emulator aware of that. `name` should be the name of the file, and `begin` and `end` should be operands that provide the address range that the file occupies. The word `TO` is optional. Without this, the debugging system will not be able to find any of the functions or global variables from those files.

The third form, where the name is preceded by a − tells the system to forget the file from its list of loaded files.

`FIND address`:

Reports the function or global variable that the given address, default hexadecimal is inside. In the case of a function, it also reports how far from the beginning of the function it is.

`FIND name`:

`Name` should be the name of a function or global variable. If it is a function, the range of addresses it occupies is reported. If it is a global variable, its single address is reported. If a same-named thing came from more than one `.obj`, `.exe`, or `.dll` it must be disambiguated using the next form of this command.

`FIND objfile/name`:

When a program `imports` one or more library, it could be that more than one function with the same name exists. Each will have to have come from a different `.b` BCPL source file of course. BCPL source files are converted into `.obj` files before they are linked together to make executable `.exe` or `.dll` files for a whole program. This form of the command only shows the addresses for the version of `name` that came from `objfile.obj`. Do not include the `.obj` suffix.

`FIND exefile/objfile/name`:

When a program imports a dynamically loaded library (`.dll`) file, it is possible that two things with the same name may have come not just from different `.obj`

files, but also from different `.exe` or `.dll` files too. This form of the command allows you to disambiguate those cases. Do not include the `.exe` or `.dll` suffix.

`FINDFILE objname`:
: `Name` should be the name of a `.b` or `.obj` file. The range of addresses occupied by objects from that file is reported. Do not include the `.b` or `.obj` suffix.

`FINDFILE exename/objname`:
: Further disambiguation, as with the `FIND` command.

## Disc access

`DISC n`:
: The emulator provides a means for examining the contents of the emulated disc drives block-by-block. If you have any discs, then disc unit 1 is automatically selected. If you have more than one disc, this command changes the disc under inspection. `n` is the unit number, default decimal.

`BLOCK n pattern`:
: `n` is the block number (counting from 0) of the currently selected disc to be viewed. `pattern` gives the format in which the contents of that block should be shown. Complex patterns should be prepared in advance, simply type them into a real unix file whose name ends with `.pat`, then `pattern` can just be the filename. Alternatively `pattern` may be one of `bytes`, `chars`, `ints`, `hex`, or `code`. Those names are translated into `512*x`, `512*c`, `128*i`, `128*w` and `128*d` respectively. Finally, if `pattern` has this form: `name=xxxx` (no spaces), then `xxxx` itself is taken to be the pattern, and it is saved under the given name so you can use it again (during the same emulator session) without having to retype it. Just use `BLOCK n name`. names may consist only of letters, digits, and underlines and are case sensitive because they refer to Unix files. If you don't want to name the pattern for future use, you must still begin it with an =, otherwise it will not be recognised.

  Patterns are (usually long) space-free strings. Actually, spaces are permitted inside quotes. They are a bit like an `out` or `printf` format string. The pattern is read character by character. Some cause something to be printed on your monitor, some cause some data to be read from the disc block under examination and displayed according to some format, and some modify future behaviour. The things that can appear in a pattern are:

  `'`anything`'` or `"`anything`"`: the anything is displayed, unprocessed, as it is.

  newline: (only allowed when the pattern is read from a file) ignored.

  a sequence of decimal digits: stored for future use as a number of repetitions.

  `*`: the next item is processed repeatedly, according to the last sequence of digits.

  `(`anything`)`: used to group patterns so that repetition can be applied to them all.

  `a`: a single byte is read and displayed as an Ascii character, invisible characters are made visible.

`c`: a single byte is read and displayed as an Ascii character, except that newlines and tabs are displayed as `\n` and `\t`, and all other invisible characters except for spaces are totally ignored.
`b`: a single byte is read and displayed in decimal, followed by a space.
`x`: a single byte is read and displayed as two hexadecimal digits, followed by a space.
`s`: two bytes are read and displayed in decimal, followed by a space.
`i`: four bytes are read and displayed in decimal, followed by a space.
`w`: four bytes are read and displayed in hexadecimal, stretched to eight digits by adding zeros to the left if necessary, and followed by a space.
`g`: a single byte is read and completely ignored.
`t`: four bytes are read and displayed as a human-readable date and time inside [...].
`d`: four bytes are read and decoded and displayed as an assembly code instruction
`_`: print a space.
anything else: printed as is, then ignored.

When the end of the pattern is reached, you are warned if the total number of bytes read is not zero, or if there were any unmatched parentheses. So, as examples:

If the entire block is just Ascii text, you would use the pattern `chars`, which is equivalent to `512*c`.

If the entire block is just normal numbers, you would use the pattern `ints`, which is equivalent to `128*i`, or `words`, which is equivalent to `128*w`.

If the block is a simple directory block consisting of an eight character file name, followed by a four byte creation date/time, followed by a four byte header-block number, all repeated 32 times (for a total of 512 bytes, a whole block), you would use the pattern `32*(8*c_ti)`.

## The interrupt system and related things

`INT n`:

`n`, default decimal, must be the number corresponding to an interrupt. It causes that interrupt to be signalled/raised.

`SHOWINT`:

The contents of the interrupt vector are shown, along with a list of the interrupts currently being processed (if any).

`INTSTATE`:

If an interrupt is currently being processed, all of the saved state, 21 words on the system stack, is shown.

`SHOWCG`:

All entries in the call gate vector (whose address is stored in the `CBGR` special register) from `1` up to the value of `CGLEN` are shown. What you see is just the hexadecimal addresses of the handlers for the system calls.

Virtual memory

`PH command`

When virtual memory is disabled (the `VM` flag is off, the default start state), the emulator treats all addresses as physical addresses, there is no other choice. When virtual memory is enabled, the emulator treats all addresses as virtual and handles the address translation automatically. Sometimes debugging requires the direct use of physical addresses. `PH` may appear as a prefix to many commands. It simply means that even if virtual memory is enabled, all addresses involved in the command are physical and will not be translated. e.g. `PH SHOW 450 TO 460`.

`VM address`:

If virtual memory is enabled, the address given, which may be any operand, is taken through the virtual memory address translation procedure, with the result of every step displayed.

The value of the page director base register is shown,
The three parts of the address, table number, page number, and offset are shown.
The table number is used to find a page table
The page number is used to find a data page and the protection bits are shown
The offset is added to find the physical address
The physical address and the contents of that memory location are shown,

`SHOWVM`:

If virtual memory is enabled, the entire contents of the page directory and all of the page tables is shown. Entries of zero are skipped. Invalid entries are shown, and marked as invalid. The entries are shown in reverse order, so that the highest addresses are at the top. For every entry in the page directory, you see its index (in decimal), the range of virtual addresses it covers, its protection bits, and "`pp N`" for the physical page number (decimal) referred to, and "`at N`" for the physical address of the beginning of that page table (hexadecimal). Indented under each of these entries, you see the same information for the contents of that page table.