15.9.1995

Computer and Automation Institute
Hungarian Academy of Sciences

Picture Manipulator & Comparator

Budapest 1995

Contents

0. Preliminaries

"You are not expected to understand this."

Caveat from the V6 UNIX kernel, documented in E. Raymond (ed.), *"New Hacker's Dictionary"*, Cambridge MA, MIT Press, 1991

This is a program to examine and modify digitized pictures (including the modification of the empty picture, that is, drawing), and to compare and discriminate different pictures (or variants of the same picture), to store and retrieve them, etc.

The program is intended to run on any PC with MS-DOS 3 or better, a 640×480 SVGA screen and some sort of mouse (actually, a three-button serial mouse was used during development, but even a two-button mouse would do). It can be run from under Windows, but does not utilize Windows' facilities and monopolizes the screen while running. It is written in C, using as few system-specific features as possible (whenever it does, Turbo C is involved), and can be adapted to other configurations in a systematic way. Nevertheless, it is parametrized to conform with *this* setup (its pixel resolution, colour generation, etc.); the present description is confined to this. Porting to other sites is dealt with in a separate Annex.

The program gets started by `picture`.

To get acquainted with program working, perhaps the best thing to do is to sit down to a PC and play around with the program while reading what follows. (A superficial acquaintance with Sect. 1 is sufficient; a more thorough reading might come after some experience with the operations themselves.) A lot of minute detail has been left unmentioned, in order to keep the exposition digestible, yet will be found out soon during experimentation. No harm can be done, except possibly by overwriting some valuable file, which, however, needs explicit action (file operations are never automatic).—As a start, hit `INS`: a small demo will be shown. Help is Ctrl+F1.

[The program is still in the process of development. Square-bracketed remarks contain alternatives, questions, suggestions about the course to be taken. Customers-in-spe are encouraged to communicate their ideas: attn. Bródy, 131–3016.]

1. Notions & Notations

1.0. *The Screen*

The screen shows (look at it) two biggish windows, the left one to be called the Yard, the right one, the Backyard.

The main operations are performed in the Yard; the Backyard has auxiliary functions (such as comparison, magnification, etc.).

These windows are bordered with "light buttons" that select colours, line types and action modes.

The bottommost line is for requests; the line above this line is for textual responses issued by the program; the topmost line contains state information.

The screen consists of pixels: coloured spots, each being in one of 16 colour states, where the 16 states can be selected from 64 possibilities. Screen size is 640×480 pixels. Pixels are referred to by screen coordinetes, denoted x horizontally and y vertically, numbered from 0 on; the origin is in the lower left corner. (The maximum screen coordinates are $x = 639$ and $y = 479$.) The Yard's origin on the screen is $29, 78$, the Backyard's is $334, 78$; both have size 296×376.

1.1. *Entities*

A *byte* is an eight-bit quantity with bits numbered 7 to 0 from most significant (leftmost) to least significant (rightmost) bit. Bytes in multi-byte quantities are numbered analogously, with higher number for more significant. (The PC stores higher bytes at higher addresses, but there will be no need to know that.)

A *number* is a four-byte (32-bit) quantity with bits numbered 31 to 0, representing a signed number in the range $[-2^{31}, +2^{31} - 1]$ with negative numbers represented in "two's complement" notation (so bit 31 is the 'sign').

A *number constant* is a (possibly signed) decimal number fitting in that range.— Wherever decimals make sense, hexadecimal numbers are also accepted; hexadecimals are prefixed with a colon (`:`), digits that are letters may be upper- or lower-case. Octals are not implemented (in retaliation for J. R. Patkul's foul execution).

There are 26 *number variables*, also called *register variables*, denoted `@a`—`@z`, letters lower-case; upper-case lettered `@`'s (`@A`—`@Z`) refer to them indirectly: take the unsigned contents of the corresponding lower-case variable mod 26 and select the variable pointed to by that number (0 is `a`, ..., 25 is `z`), that is, the variables act as an array, indexed by any of them.

Numbers can be 'stacked', and stacked numbers may serve as additional number variables, called *stack variables*, denoted `$`*n* and `&`*n*, with *n* a number constant: the location counted from stack top. The stack area can also be used as plain variable area: `$^`*n* and `&^`*n* count from stack bottom.

A *point* or *coordinate pair* is a pair of numbers (constants or variables), representing a pixel on the screen.

A *text* is a (possibly empty) string of (non-control) characters, not exceeding 80 characters.

A *text constant* represents a text; its internal coding uses two bytes per character, appended with a (two-byte) zero.

There are 10 *text variables*, denoted `@0`—`@9`, each taking a text, represented as above.

A *name*, or *identifier*, is a text of up to 12 characters, starting with a letter, followed by (zero or more) letters or underscores (`_`). (No digits.) (Note that this notion is different from what is usually called 'name' under the operating system; nevertheless, DOS file names will also be called so; it will be clear which is which.) Names (in the present sense) are used to identify (customer-defined) routines and labels. By convention, upper- and lower-case letters are considered no different. A name may be longer than 12 characters but only the first 12 are significant. (A name, being a text, may also be put into a text variable.)

The program executes *requests*: commands, possibly with arguments.

Requests may come either from the *console*, meaning the keystrokes of the Human Interactor sitting before the screen, or from *request files*, meaning prepared and stored

request sequences activated from the console or another file. Request files can be attached / changed / dropped (by requests) at any time. Console input and request-file input can be freely intermixed.

Request sequences can also be composed into *routines*, again possibly with arguments. A *library* is a prepared and stored collection of routines. Libraries have to be attached to the program to make the routines in them accessible; library attachment is also changeable at any time. Routines, as seen from the customer, act as ordinary requests. They can be activated both from console and file.

1.2. *Colours*

Colours are defined in the `r`(ed-)`g`(reen-)`b`(lue) system; each component may have four intensity levels, denote them 0–3 from absent to maximum brightness; so 64 colours are definable. In particular, a colour with $(r, g, b) = (0, 0, 0)$ produces black, $(3, 3, 3)$ produces white.

The complement of colour *col* with definition (r, g, b), denoted `cpl`(*col*), is the colour with definition $(3 - r, 3 - g, 3 - b)$. They are in fact complementary in the sense that their `or` is white and their `and` is black. Still, the colour distribution is rather uneven, and it may take quite a bit of experimentation to mix a colour that is somewhat similar to the desired one, or to find well-contrasting colours.

At any time, any 16 of the 64 definable colours can be shown; such a set of 16 colours is called a *palette*; palettes can be specified / changed run-time.

The component-wise definition of colours is used only within palette specifications; once a palette is specified and activated, all screen operations refer to colours by their location in the palette, a number ranging from 0 to 15 (`:0`–`:F`), the *colour number*. (So the same colour number may denote different colours in different palettes.)

Between colours, logical operations (`and`, `or`, etc.) can be performed, but they operate on the colour numbers (rather than the colour definitions), so their outcome can be quite unrelated to the optical properties of the colours involved.

8 palettes are available, numbered 0 to 7. [There could be more.]

Palette 0 is the "official" palette, the one that is specified and activated by the operating system.

Palette 1 is a setup that has been felt fit to be the basic working palette for this program; colours in it are ordered approximately according to overall brightness, and colours with numbers n and $15 - n$ (that is, colours complementary in the sense of operations on colour numbers) are also complementary in their definitions; moreover, the palette tries to include most of the (few) non-disgusting colours.

Palettes 2–7 initially equal palette 1; they are intended to be freely redefinable. (0 and 1 can also be redefined, but don't do that.)

Initially, palette 1 is activated.

The Yard and the Backyard have a ground colour, `g`, initially white (more accurately: colour number 15); redefinable.

At any given instant, there is an actual drawing colour, `c`, initially black (number 0); redefinable.

The `xor` of the (numbers of) `g` and `c` is used in many operations, to be denoted `x`.

1.3. *Lines*

There are three types of lines ("strokes") used in drawing.

Type 0 – width is one pixel, contiguous pixels meet in an edge or a vertex;

Type 1 – width is one pixel, contiguous pixels meet in an edge;

Type 2 – width is three pixels, contiguity structure is determined by the requirement that there must be three pixels in at least one axis direction (the one "more perpendicular" to the local line direction); line ends are either *cut* (again in the "more perpendicular" direction) or *bloated*: having a 3×3 blot *around* the nominal endpoint.

Lines are symmetric for all mirror symmetries of the square around the starting (or ending) point (meaning that lines in mirror positions are mirror images of each other), with the proviso that $\pm45°$, $\pm135°$ count as "more horizontal than vertical", which makes no difference for type 0. Lines of type 0 and 2 are the same if drawn in the opposite direction, those of type 1, not necessarily.

A line of length 0 appears as a point: with types 0 an 1 a single one, with type 2 (even for a cut line) as a 3×3 blot.

1.4. *Characters*

Text handling comprises very different and physically unrelated actions: Text may come from the console or from files, it may go into a file or onto the screen, become, or not, part of a picture, etc.

At output (on the screen), the shape of characters depends on the current *font*.

Eight character fonts can be defined, they are numbered 0 to 7.

Presently two character fonts are available: a bigger one (8×15 pixels), font 0, and a smaller (8×8), font 1. (Accents may add to the vertical dimensions.) (For more detail, cf. Sect. 2.10.)

[At least one more font will be added: a small sans-serif font (variable-width$\times 7$, with digits and upper-case letters not exceeding 5 pixels vertically).]

There is a concomitant character-design program to produce new fonts or redesign characters in existing ones, so customers are not confined to what they are presented with.—While *existing* fonts can be changed run-time, font *redefinition*, unlike palette redefinition, requires program recompilation. [This restriction can be raised.]

Every communication with the program appears in font 0; other fonts are only for application within pictures.

The internal character representation of the program, to be called the *concise* representation, takes a two-byte (16-bit) unit to hold a character. Most input sources will not be able to directly produce this representation, so the program understands another one, call it *verbose*, that takes a byte per character (or the lower byte of a two-byte unit, with high byte zero) and uses exclusively the 'printable' ASCII character set (the characters with codes `:20`—`:7E`) and a 'carriage return' or 'newline' (with code less than `:20`). A lot of characters that take a single unit in concise are expressed by a sequence of characters (an 'escape sequence') in verbose.

The printable ASCII characters are contained both in concise and verbose, and have their ASCII codes, except the characters `!` (exclamation mark) and `‘` (back-apostrophe) have special functions: they serve as escape characters. In verbose, to get an `!` or `‘`, type `!!` or `!‘`, resp. In concise, the 'true' `!` and `‘` are coded with high byte set to `:80` (that is, `:8021` and `:8060`, resp.).

In extension of the ASCII printable character set, 32 additional character patterns can be included for character codes `:00`—`:1F`. By definition, the character with code `:7F` equals that with code `:00`.

These additional characters are intended to produce arbitrarily designed visible patterns on the screen. A tentative set of such characters has been implemented as shown below; any other set would do, even different ones in different fonts. Note, however, that three codes have predefined usage: the program uses character `:00` as "cursor", and characters `:1E` and `:1F` for 'pointless' `ı` and `ȷ`.

(As a matter of fact, any character can be redesigned in any way; still, the suggestion is to adhere to ASCII for codes :20—:7E, and restrict arbitrariness to :00—:1F.)

In concise, the additional characters are represented by their codes. In verbose, they are referred to with !-prefixed symbols; letters in upper-case and lower-case are considered different. [Naming conventions can also be changed.]

:00 | !* – ■ (all-black),
:01 | !q – □,
:02 | !l – ◇,
:03 | !R – →,
:04 | !U – ↑,
:05 | !L – ←,
:06 | !D – ↓,
:07 | !C – ∉,
:08 | !P – £,
:09 | !Y – ¥,
:0A | !S – §,
:0B | ![– «,
:0C | !] – »,
:0D | !E – ∈,
:0E | !~ – ≈,
:0F | != – ≡,
:10 | !< – ≤,
:11 | !> – ≥,
:12 | !+ – ±,
:13 | !: – ÷,
:14 | !x – ×,
:15 | !. – ·,
:16 | !r – √,
:17 | !s – ∑,
:18 | !p – ∏,
:19 | !u – ∪,
:1A | !n – ∩,
:1B | !I – ∞,
:1C | !0 – ∅,
:1D | !o – °,
:1E | !i – ı (for Turkish or with accents),
:1F | !j – ȷ.

The colour of the characters can be controlled in a similar way. Normally, characters start to appear in the 'current drawing colour'. A 'colour switch' character can appear anywhere in the character string, it changes the colour of subsequent characters to the colour encoded in the switch. In concise, the code is :20 in the low byte, :40 or'ed with the colour number in the high byte. In verbose, the coding is ! followed

by the colour number given as a hexadecimal digit; the digit must *not* be preceded by a colon, and if it is a letter-digit: `a`–`f`, it must be lower-case.

Any *other* character preceded by an `!` appears in concise with its low byte unchanged, high byte set to `:80`.

There is a set of 16 accents ("diacritical marks"). Every visible character can be accented. In concise, the high byte of the accented letter takes `:80 or`'ed with the accent number. (`:80` is not mandatory, except for the 'empty accent'.) In verbose, accents are `'`-prefixed symbols; the accentuation has to *precede* the character. [Accents and their names can be changed too.]

`:0`	`'␣` (`'` followed by a space) – the "empty accent", e.g., ı,
`:1`	`'.` – e.g., ż,
`:2`	`'\'` – e.g., í,
`:3`	`'`` – e.g., ì,
`:4`	`'"` – e.g., ï,
`:5`	`'^` – e.g., î,
`:6`	`'H` – e.g., ő,
`:7`	`'~` – e.g., ñ,
`:8`	`'=` – e.g., s̄,
`:9`	`'u` – e.g., ĭ,
`:A`	`'v` – e.g., č,
`:B`	`'o` – e.g., å,
`:C`	`'c` – e.g., ç,
`:D`	`'s` – e.g., ş,
`:E`	`'d` – e.g., ḥ,
`:F`	`'b` – e.g., ṯ.

Ordinary 'i' and 'j' can go accented, the dot on top is automatically discarded for upper accents.

(All this is very similar to TEX's ways, except no braces are ever used, and two 'escape characters' are appointed—neither of which is `\`, so it's easier to put them in C "strings".)

The keyboard has a lot of "control keys", which are widely used to start program actions, not always strongly related to the markings on the keys.

In concise, they get unorthodox codes, with bit 7 set. In verbose, they are written as three-character tokens, upper-case, prefixed with a point (period, full stop). (Writing the control-key symbols in character form doesn't make much sense in console input but enables producing 'keystrokes' from routines and request files in verbose.)—There are no settled conventions for naming the control keys. The following identification seems unambiguous for most keyboards I've come across. In the sequel, the verbose form will be used to name the keys / combinations in the text of this description.

:81 | .ETR – the key called 'Enter' (or 'Carriage return' or 'CR'),
:82 | .BSP – backspace (often marked with a back-arrow but is not an 'arrow key'),
:83 | .ESC – 'Esc', escape,
:84 | .TAB – 'Tab', tabulator,
:90 | .INS – 'Ins', insert,
:91 | .DEL – 'Del', delete,
:92 | .HOM – 'Home',
:93 | .END – 'End',
:95 | .PGU – 'PgUp', page up,
:96 | .PGD – 'PgDn', page down,
:97—:9A | .RAR, .UAR, .LAR, .DAR – the arrows →, ↑, ←, ↓,
:A1—:AA | .F01—.F10 – the numbered functional keys (some keyboards have keys up to 12 or 16; they remain unused).

Keys are often to be depressed together with one or more of 'Shift', 'Ctrl', 'Alt'. In concise, this is indicated by `or`'ing the character code with `:0100`, `:0200`, `:0400`, resp. In verbose, this is written by setting '`s`', '`c`', '`a`', lower-case, after the point, before the three-letter token (in any order if more of them).

(Not all modifier / control key combinations can be obtained from the keyboard—some are confiscated / disconfigured by the authorities. Those used in the program do exist everywhere.)

From the console, control-key input can be produced directly. The modifiers `s`, `c`, `a` are `or`'ed onto the code of all characters (not only control keys) they modify, except `s` is not shown in the high bytes of codes for 'ordinary ASCII' characters (they being different anyway). The 'arbitrary' characters and the colour switches need verbose coding, as well as most of the accents, except that for the "Hungarian" accents the following shorthand is available with console input (but not with input from a file): simultaneously holding (depressing) 'Ctrl' yields 'long' (ó), 'Alt' yields 'umlaut' (ö), 'Ctrl' *and* 'Alt' yields 'long umlaut' (ő). (On the other hand, the 'HUNKEY' convention is not supported; it has been felt that the character positions usurped by it are going to be much heavier used in their original meaning.)

Request files and library files are read bytewise; they have to use verbose coding; all characters outside the range `:20—:7E`, e.g., carriage return or newline, convert to `.ETR`, a sequence of them, to a single one. Request files have the additional control symbols (not corresponding to keystrokes) `.FIN`, `.BRK`, `.ECX`, `.CYC`, `.LVC`, `.RPT`, coded `:F0—:F5`; library files have, moreover, `.RET`, `.XIT`, coded `:F6—:F7`.

1.5. *Mouse*

A three-button mouse is assumed. (No problem if otherwise: button clicks can be substituted by keystrokes or other means, cf. Sect. 2.4.)

Note that to activate the mouse, or all three buttons, some systems and some mouses require the left button, or all buttons, to be held down during power-up. (Reset without power off / on is not enough.)

2. Operations

2.0. *General*

Operations are activated by mouse clicks or request lines.

A request line can come from the console or a file. It consists of zero or more request sentences, terminated by depressing a control key (or by giving, in character form, its symbol). In particular, a control key in itself is a 'request line'.

If there are several request sentences in a line, they are to be separated by semicolons (;). The line must not exceed 80 characters.

Execution starts when the terminating control key is given; request sentences are executed in order, at last the action of the control key is performed.

If no special control key action is needed, terminate with `.ETR`, or, in a file, by starting a new line. On the other hand, no new line is required in a file to terminate a request line when it is terminated by other means.

If the line becomes full in lack of a terminator, it gets terminated with `:00FF`.

A request sentence consists of an operation code and possibly arguments to it; arguments are, depending on the operation, particular signs, number or text constants or variables, etc. A routine call is also considered a request sentence; in this case the operation code is the routine name.

Letters in operation codes can be upper- or lower-case. Arguments are separated by spaces or commas (,). Empty request sentences, as well as redundant spaces around codes and arguments, are permitted.

With console input, the request line, as long as not terminated, can be edited in a rudimentary way: `.BSP` backsteps (deletes last characters) one by one, `.ESC` discards the whole lot. `.BSP` and `.ESC` must be the control keystrokes themselves (not their character symbols).

Any tool can be activated at any time, except the mouse is inactive (and invisible) when a request line has been started and is not yet finished (or discarded).

Some operations elicit textual responses or send messages, appearing in the response line. Such responses remain visible until the next response or message, or until the end of the next request line, whichever comes first.

2.1. *Miscellaneous*

`.END`

Terminates program run.

`.DEL`

Restarts the program, restoring initial values, except attached request files / libraries remain attached.

`ERR` *c*

Sets error notification condition: requests in error appear in the response line and suspend program execution—for bit 0 set if coming from console, for bit 1 set if coming from request file. To skip, strike `S` or `s`: the request will be forgotten and execution resumes; any other key terminates program run. Errors not to be notified cause a short beep, get skipped, and run continues. Undefined control keys act as `.ETR`, adding only the beep (no notification is given). Initial value is 3.—Structural errors in library routines get notified unconditionally; the choice here is between skipping the culprit or exiting the library call altogether.

`.cF07`

Shows (coherent) date and time; date also goes to `@x` as days since 15 Oct. 1582, time to `@y` as hundredth of seconds since midnight.

`.cF08`

Converts the contents of `@x` (interpreted as with `.cF07`) to year / month / day, and puts year in bytes 3–2, month in byte 1, day in byte 0 of `@y`.

`.cF09`

Informs of the free space available (for library and request-file attachment and picture saving); space is given in bytes. Space goes to `@x`, the number of pictures that can be saved, to the low byte of `@y`.

`SYS` *command*

Calls DOS to execute *command*, which may be anything it can execute (including the running of programs). Normal screen handling is *not* restored, so if DOS is going to write something, this will be illegible, and will, moreover, destroy screen contents.

`SYST` *command*

The same as `SYS`, except the screen is saved / restored and DOS finds its normal working conditions (those that prevailed before the start of `picture`).—Screen saving

requires as much space as saving *two* pictures. If less is available, it asks the console whether to go on or not; if going on, it will orderly execute but the Backyard, and possibly also the Yard, will be empty at return.

`SYSTEM` *command*

The same as `SYST`, except it waits for (any) keystroke after returning from DOS and before restoring the screen.

`%` *anything not containing a* `;`

Comment, extending till the next `;` or (if no `;`) till the terminator; the terminator remains functional (is not "commented out").

`.cF01`

Help, offered in the Backyard. Requires the file `picture.hlp` to be present in the DOS directory where*from* `picture` is being run. (When `picture` gets renamed, the help file has to be renamed the same way, appended with the extension `.hlp`.)—The program saves and restores the Backyard; if unable, asks the console as to what to do.

2.2. *Numbers & Texts*

Number Operations:

Recall that number constants must fit into the range $[-2^{31}, +2^{31}-1]$, and are written either as possibly signed decimals or as hexadecimals prefixed with a colon.

Number variables are initially 0. `@x`, `@y` and the low byte of `@z` are also set by some non-arithmetic requests; the higher bytes of `@z` are extensively used to convey program control information.

There are two number stacks, each 256 deep: one for the use of library routines, called the 'library stack', one for everything else, call it the 'plain stack'. Routines cannot push / pop on the plain stack, nor non-routines on the library stack, but otherwise both stacks can be accessed from everywhere. `$`n refers to elements on the plain stack, `&`n, to those on the library stack, with n a number specifying the position of the element in the stack, stack top being 0; `$^`n and `&^`n count the position from the stack bottom.

In what follows, *variable* will mean any of the `@a`–`@z`, or a reference to them by `@A`–`@Z`, or `$`n, `$^`n, `&`n, `&^`n.

The push / pop opeartions are as follows. The stack (plain or library) affected by them depends on the caller. Causing overflow or underflow counts as 'error'.

`+` x or $\uparrow x$

x is a number constant or variable: it gets pushed onto the stack. Bit 8 in `@z` is set if successful, cleared if not (stack full). When `@z` is being pushed, it is first pushed, then set. If x is relative to the top of the same stack that is being changed, the reference relates to the pre-change state.

`-` x or $\downarrow x$

x is a number variable: the stack top gets popped into it. A reference to the same stack again relates to the pre-change state (a reference to the stack top is perfectly legal, but the popped contents will be lost).

`~` x

x a number constant or variable: the stack pointer gets changed by that amount. The elements gained by increasing the pointer have indefinite contents (remnants of previous stack operations).

`.cF06`

Informs of the stack load (= number of elements stacked), plain stack going to `@x`, library stack, to `@y`.

Arithmetic:

x `op` y

x is a number variable, y is a number constant or variable, `op` is one of the following.

`=` – the value of y to go to x;
`+=` – $x + y$ to go to x;
`-=` – $x - y$ to go to x;
`*=` – $x \times y$ to go to x;
`/=` – $\lfloor x/y \rfloor$ to go to x ("integer division": $x \div y$);
`%=` – x mod y to go to x ("modulo");
`&=` – $x \wedge y$ to go to x ("`and`": bitwise mod 2 multiplication);
`^=` – $x \oplus y$ to go to x ("`xor`": "exclusive or", "antivalence", "symmetric difference", bitwise mod 2 addition);
`|=` – $x \vee y$ to go to x ("`or`": "inclusive or");
`<<=` – $x \times 2^y$ to go to x ("left shift");
`>>=` – $\lfloor x \times 2^{-y} \rfloor$ to go to x ("arithmetic right shift", the sign bit is shifted in).

With shifts, a single `<` or `>` is sufficient.

Second operands in shifts are considered unsigned numbers. (With `and`, `xor`, `or`, number type is of course irrelevant.)

Overflow simply overflows; division by zero yields 'signed infinity' (that is, the number with the appropriate sign and max. abs. value). Division rounds in abs. value. (So $(-3) \div 2$ is -1, rather than -2: instead of $\lfloor x/y \rfloor$ above, $\mathrm{sgn}(x/y) \cdot \lfloor |x/y| \rfloor$ should have been written.) Modulo adheres to the "mathematical" convention: there are n residue classes mod n (and, e.g., (-1) mod 3 is 2). mod 0 yields 0.

-2^{31} (= `:80000000`) makes no sense in certain signed-arithmetic operations since it has no negative within the value range.

[Integer square root and vector length ("Pythagorean addition") to be added?]

n

n a number constant: abbreviation for `@x =` n. (Do *not* use a `+` sign; `-` is OK.)

n, m

n, m number constants, as above: abbreviation for `@x =` n`; @y =` m.

x `?`

Shows the contents of x in the response line.

Text Operations:

Text constants, as used in text operations, are terminated either by their 80^{th} character if exceeding that length, or by the end of the request sentence: the semicolon or the terminator. To have `;` in a text constant, give `!;` (to have `!;`, give `!!;`).

Text variables @0–@9 are initially empty. @9 is also set by some lettering requests.

Texts (constants or variables) may contain references to variables. The 'expansion' of a text is the text with the variable references in it replaced by the current contents of the variables (in decimal if numeric). Expansion is a single-level process: references popping up while expanding a reference are not further expanded. @@, $$, && expand to @, $, &, resp. (Thus, e.g., @@0 expands to @0; a subsequent expansion will replace this @0 with its contents.) To get a @, $, & that is never expanded, write !@, !$, !&.

u = '*v*

u a text variable, *v* a text constant: the expanded text of *v* goes to *u*. (A single quote (apostrophe) is used. Notice that there is no closing quote. If there are subsequent quotes, they belong to the text.)

u = *v*

v a text variable: the text *in v* gets expanded, otherwise as above.

(Notice the subtle difference between, e.g., @0 = @1 and @0 = '@1: if @1 contains, say, @a, then @a gets expanded in the first case and is copied as it is in the second.)

u ~ '*v*

u a text variable, *v* a text constant: compares *u* unexpanded with *v* expanded lexicographically (according to the character codes) and sets the low byte of @z to 0, :FF, 1 for 'equal', 'the first precedes (is smaller than) the second', 'the first follows (is greater than) the second', resp. Only the low bytes are compared (so, e.g., accents are neglected); colour switches are disregarded.

u ~ *v*

v a text variable.

u ?

u a text variable: shows its contents, character length and pixel width; length and width go also to @x and @y, resp. Contents are shown unexpanded, but length and width are given for the current expansion and in the current font.

'*u*

u a text constant: abbreviation for @9 = '*u*.

"*u*

u a text constant: message, to appear in the response line.

(For using texts in picture lettering, cf. Sect. 2.10.)

2.3. *Conditionals*

Conditional sentences affect the remainder of the request line: if they evaluate true, the remainder gets executed, if not, not—the line acts as if terminated with `.ETR` in place of the conditional. (The remainder of the line may contain further conditionals.) (Conditionals, too, need a semicolon separator if not immediately followed by the terminator.)—Conditionals have the form

`cond` x, y

where x and y are number constants or variables and `cond` is one of the following: `IFEQ`, `IFNE`, `IFGT`, `IFLE`, `IFSET`, `IFCLR`; the condition is true if $x = y$, $x \neq y$, $x > y$, $x \leq y$, some of the bits set in y are set in x, none of the bits set in y is set x, resp., false otherwise. With a zero second operand, `IFSET` is always false, `IFCLR` is always true.

[Instead of "rest of line", parenthesizing might be introduced.]

(Nobody will ever be tempted to use conditionals from the console; they are intended for use in request files and routines, cf. Sects. 2.17–18.)

`.sF04`

Waits for a (single) keystroke from the console (*always* from the console, even if input is coming from a file), and puts its 'concise' code in `@x`. The incoming keystroke has no other effects (not even if a control key).

(This is very obviously *not* a 'conditional'. But I couldn't find any other place to mention it, and its most natural application is, after all, in asking for decisions.)

2.4. *Mouse Handling*

Mouse positioning can be done (as if by magic) manually—in case program control is with the console. (Otherwise, manual motion has no effect.) Mouse buttons are also inactivated while being in the middle of a request line (even if given from the console).

= x, y

Sends the mouse to x, y, always effective.

= in itself (without arguments)

Shows, in the response line, the position of the mouse and the colour of the pixel at that location and sets `@x`, `@y` to the location and the low byte of `@z` to the colour.

`.sF01`, `.sF02`, `.sF03`

Equivalent to click left, middle, right button, resp., always effective.

`SPEED` n

Sets the mouse speed to n "mickeys", $1 \leq n \leq 100$; initially 50. (Ineffective with some mouse drivers.)

`SPEED` in itself (without argument)

Shows the mouse speed in the response line. (Falsely shown by some drivers even if setting is effective.)

`.sPGD`

Hides the mouse symbol; a second `.sPGD` displays it again.

`.sPGU`

Changes the symbol into a magnifier (loupe); a second `.sPGU` undoes the change.

2.5. *Colour Selection*

/ *pal*

Selects the palette *pal* ($0 \leq pal \leq 7$). The initial palette is palette 1.

`.cF03`

Shows the current palette in the Backyard.

\ *pal*, *col*, *r*, *g*, *b*

(Re)defines *col* of *pal* to have component brightnesses *r*, *g*, *b* ($0 \leq pal \leq 7$, $0 \leq col \leq 15$, $0 \leq r, g, b \leq 3$). The new setting takes effect when the palette gets activated *next* (even if it is the current palette that is being modified).—The program expects that, in every palette, colour 0 is 0,0,0 (black) and colour 15 is 3,3,3 (white). (This is not enforced, but the screen may look funny otherwise.)

`.F10`

Complements (the definition of all colours in) the current palette; the effect is immediate. (Some picture details may become more discernible.—A second `.F10` obviously reverts to the original.)

`N` *col* 1, *col* 2, *col* 3

Sets the 'notification' colours: *col* 1 is the colour of the response-line texts and of the frames of the selected light-buttons, *col* 2 is the colour of the request-line cursor and of pixel-boundary marking in grid-magnifications, *col* 3 is the colour of the clock-time display. Initially 10, 8, 4, resp.

`G` *col*

Selects *col* as the background colour ($0 \leq col \leq 15$). It is initially 15. (Changing it does not affect anything that is currently visible; the effects start with subsequent operations.)

`H` *col*

Selects the halting colour; initially 0.

`C` *col* or any mouse button or the horizontal arrows `.RAR` / `.LAR`

Selects the drawing colour; initially 0.

`X` *col* 1, *col* 2

Changes pixels of *col* 1 to *col* 2 in the Yard.

`Y` *col* 1, *col* 2

Mixes pixels of *col* 1 with *col* 2 in the Yard: *col* 1-pixels having even coordinate sum turn to *col* 2.

`Z` *col*

Blinks pixels of *col* on the screen (not only in the Yard); until the next keystroke if the request is coming from the console (the character gets discarded), for 1 sec otherwise.

`.cF05`

Counts the pixels in the Yard having the current drawing colour, and puts the count in `@x`.

2.6. *Line Type Selection*

S l or mouse buttons

Selects the line type (stroke) l ($0 \leq l \leq 2$); the line-type symbol in the upper left corner gets changed by the mouse: the left button increases, the right button decreases the type cyclically. Initially, the type is 1.

2.7. *Action Mode Selection*

Action modes specify the effects of mouse button clicks (and their control-key and symbol equivalents) when the mouse is pointing into the Yard or the Backyard. (They don't affect mouse operations when the mouse points outside, nor non-mouse operations.)

Action modes are represented by the symbols in the vertical array at left; they are numbered 0 to 15 from bottom to top.

`M` *mode* or any mouse button or the vertical arrows `.UAR` / `.DAR`

Selects the action mode; initially 0.

2.8. *Drawing*

`.F01`

Clears the Yard, that is, sets it to the ground colour `g`.

`.F02`

Clears the Backyard to `g`.

Drawing is done in the Yard or in the Backyard, whichever the mouse happens to point to, on whatever happens to be there: nothing, result of previous operations, pictures coming from extraneous sources, etc. The reference point is the pixel pointed to by the mouse. If the mouse is outside those regions, nothing is drawn, and no drawing can continue outside the region it started in (no harm is done in trying, and drawing gets resumed when returning to the region: drawings get "clipped").

Drawing starts from the reference point. Drawing can be done either by manually moving / clicking the mouse—cases (i)–(iii) below—or by request sentences—case (iv). While drawing manually, mouse clicks cannot be replaced by control keys or their symbols. (Evidently, files can only draw with explicit requests: case (iv).) While the mouse symbol is hidden or is a magnifier, manual drawing is disabled. [This can be reconsidered.]

The current line type is used.

Colouring has the following variants.

Paint: set the colour of the affected pixels to `c`, the current drawing colour.

Stain: set the colour of the pixels to `cpl(x)` (recall that `cpl` is the complement, and `x` is the `xor` of `c`, the drawing colour, and `g`, the background colour).

Mark: `xor` the colour of the pixels with `x` itself.

Erase: set the colour to `g`, the background colour.

Notice that marking is a *reversible* operation, meaning that executing it a second time undoes the effect. (Erasing does not necessarily revert the effect of drawing: drawing may have changed pixels that have had colours different from the background.)

The following operations are available.

(i) *Select mode* 0:

A line (curve) is drawn along the path of the motion of the mouse as long as a button is being held down.

The left button paints, the middle button stains, the right button erases. Erasing is always done with line type 2 (irrespective of the current type).

Lines of type 2 are bloated.

(ii) *Select mode* 2:

Draws a straight line segment, starting at the location of the first click and ending at the second one (showing in the meantime a segment connecting the starting point and the current mouse location).

The left button paints, the middle button marks, the right button *drops* the segment (instead of erasing anything). It is the second click that decides upon what happens (the first click can be done with any button).

Lines of type 2 are cut.

(iii) *Select mode* 3:

Like (ii) but draws a rectangle with one vertex at the first click and the opposite vertex at the second click.

Lines of type 2 put the fat on the outside.

(iv) *Drawing requests* (acting irrespective of the current mode setting):

`P`

Paints a point at the current mouse location x, y.

`Q`

Like `P` but marks.

`L` dx, dy

Paints a straight line segment connecting the mouse location x, y with $x + dx, y + dy$, endpoints included.—`L` $0, 0$ is a point.

Lines of type 2 are cut.

`K` dx, dy

Like `L` but marks.

`A` r, f, t

Paints a circular arc counterclockwise around the mouse location, with radius r, starting angle f and ending angle t, f and t are in degrees.

[This is not yet really implemented, since it uses the "official" arc drawing facilities that have a lot of shortcomings. Moreover, the form of the request itself ought to be reconsidered: Preferably, the current location might be the starting point of the arc (rather than its centre), and, instead of the radius, an arc definition based on a Sabin-type "bulge" might be introduced (which, in contrast to the definition involving the

radius, is numerically more stable, and enables, within the number representation, the definition of *all* arcs that are drawable). Instead of "exact" interpolation, a Bernstein-type approximation could be implemented (excellent!).]

`E` r, f, t, d

Paints an elliptical arc, defined as a distorted circular arc: the horizontal extension of the circular arc as given by the first three arguments is unchanged, its vertical extension is magnified by d, given in percents (100 means undistorted).

[The same reservations apply as to circular arcs.]

`B` bx, by, cx, cy, dx, dy

Paints a third-degree Bernstein arc ('Bézier spline') connecting the mouse location x, y with $x + dx, y + dy$, the 'control points' having coordinates $x + bx, y + by$ and $x + cx, y + cy$. (Again, `B` with all-zeros is a point.)

Lines of type 2 are cut.

(There is a "tutorial" request file `bezier.prq` to experiment with such arcs.)

(Definitions of Bernstein arcs and chains of them will probably seldom if ever generated manually. To get a taste of their application for similar purposes, the best source I know is D. E. Knuth's "Metafont", where, among other things, methods of automatic control-point generation are discussed. Incidentally: Metafont's "tracing" capabilities enable the production of character-format files wherefrom Bernstein arc / chain definitions can be generated for the present program.)

[Or: the methods can be stolen and implemented directly. Since, in the currently planned applications, Bernstein arcs are likely to be applied in producing some basic pictures but not in the everyday manipulations on them, it is, at least, not urgent to incorporate those methods in the program itself.]

`O` i

If $i > 0$, the parameter space is to be partitioned into i equal-length intervals while generating the *next* Bernstein arc; $s \leq 0$ leaves alone / restores ordinary working. (Normally, the program chooses its steps automatically; *this* can be overridden by `O` for a *single* arc. `O` i might improve, or not, upon smoothing; on the other hand, small values of i produce a *polygonal* arc with vertex points on the Bernstein arc. High values may cause overflow and draw mock Chinese characters.)

[A good question: Should `L`, `K`, `A`, `E`, `B` reposition the mouse into the ending point?]

2.9. *Tagging*

The symbols of modes 4–11 are called *tags* (or *markers*).

When in one of those modes, a mouse click puts the symbol into the Yard or Backyard; the midpoint goes to the reference point.

With line type 0 or 1, tags are drawn as shown; with type 2, strokes are tripled.

Left paints, middle marks, right erases.

[Tag symbols can be redesigned like characters.]

2.10. *Lettering*

A character has a width w, a height h and a depth d; w may be different for different characters, h and d are defined the same for all characters of a font. ($w \times (h + d)$ is a sort of enclosing rectangle; the 'black' of the character needn't span the whole of its measurements.)

In horizontal mode, when a character gets placed to a pixel position, its h extends upwards from that pixel, including that pixel; d extends downward, excluding the pixel; w extends to the right, including the pixel. (So the rectangle of a character placed at x, y extends over the horizontal pixels x to $x + w - 1$ and the vertical pixels $y - d$ to $y + h - 1$.)

Upper / lower accents may add to the vertical sizes, this, however, is not a fixed amount: accents are placed higher or lower according to the 'black' of characters.

In vertical mode, read left for upward, right for downward, upward for right.

Only the pixels to hold the 'black' points of characters get recoloured in the picture, the rest remains untouched.

Font 0 has $w = 8$, $h = 12$, $d = 3$; accents add either 4 pixels to h or 3 pixels to d in the worst case.

Font 1 has $w = 8$, $h = 7$, $d = 1$, with accent excesses 3 / 2.

[The planned font 2 will have w variable-size, $h = 5$, $d = 2$.]

(This is not the whole story; characters have a lot of other parameters: e.g., "step", which equals w for the fonts above but can be different, etc. For more detail, consult the character-design program.)

* *f*

Sets the font ($0 \leq f \leq 7$). [Only 0 and 1 make sense now.]

`.cF04`

Shows the current font in the Backyard.

_ '*text*

Puts the expanded text constant into the picture, writing horizontally, starting from the pixel pointed to by the mouse.

| '*text*

The same but vertically.

In modes 12 and 13, texts are inserted in the Yard or Backyard at mouse clicks: horizontally for 12, vertically for 13.

A mouse click prompts for inputting the text, in the request line, terminated with `.ETR` (or any other terminator); editing is possible as long as unterminated; the text goes to the picture after termination. No quote is to be given at the beginning, and a semicolon can be written as it is (without `!`-prefix). The text gets expanded.—It will be painted if the button was the left one, marked if middle, erased if right.—The text going to the picture goes also to `@9`. If `@9` is referred to, it is first used, then overwritten.

[Should the mouse be shifted to the position following the end of the text?]

2.11. *Filling* (*Flooding*)

Filling is done in mode 1.

It starts from the Yard or Backyard pixel pointed to by the mouse.

Two methods are available.

(i) *Filling proper*: All neighbours and neighbours-of-neighbours that have the same colour as the starting pixel get recoloured (any colour different from it, including the filling colour itself, bars colour propagation).

(ii) *Flooding*: Everything gets recoloured until neighbours having the halting colour `h` stop the process.

In both cases, neighbours are the pixels that share an edge (so horizontally and vertically adjacent pixels are neighbours, diagonally adjacent ones are not).

(i) is soft-coded and comparatively slow. (ii) is an "official" benefit, quick but defective: it does not correctly flood regions that have complicated shapes, cannot be orderly clipped, and is able and willing, in some circumstances, to bring the computer to a standstill. (It is real fast and it is often handy, so it has been included in the program. But be careful to minimize losses when trying it under previously untested conditions.)

Left button starts (i) with colour `c` (it "paints").

Middle button starts (i) with `cpl(x)` (it "stains").

Right button starts (ii) with `c`.

(i) can be stopped in midtime by hitting any key (the character gets thrown away).

(Notice that, as a consequence of the definitions, lines of type 1 and 2 can themselves be filled, lines of type 0 not necessarily—in fact only if they are composed of axis-parallel segments; filling is revertible / recolourable if and only if the filling colour was different from the limiting colour(s).)

2.12. *Magnification & Pixelwise Modification*

In modes 14 and 15, (clippings from) the picture in the Yard can be magnified into the Backyard; pixel alterations in the magnified picture modify the original.

Mode 15 shows the pixel boundaries, mode 14 does not.

`U` m

Sets the magnification, $2 \leq m \leq 32$ (can be 1 in mode 14).

If the mouse is in the Yard, a click (any button) magnifies its surrounding; if it is in the Backyard, left paints, middle stains, right erases the pixel; immediately clicking again at the same pixel reverts to the previous state (the state "alternates").

2.13. *Filtering*

Filters operate in the Yard.

Two kinds of filters are presently available, both perform "contour" operations:

`.F08` and `.F09`

They retain the boundary pixels of the picture and clear (set to g) the rest. A boundary pixel is a pixel that has a neighbour of a colour different from its own. For `.F08`, neighbours are the pixels that meet in an edge (that is, the four pixels that are horizontally or vertically adjacent); for `.F09`, pixels meeting in a vertex (that is, the four diagonally adjacent ones) are also neighbours. At the Yard's periphery, there are *no* outside pixels. (More accurately: Imagine the Yard surrounded with replicas of its horizontal and vertical edges, joining in the replicas of its vertices, and work in that enlarged area. This obviously is an unambiguous construction, yields an unambiguous decision for the pixels inside, and amounts to what has been said.)

[Other filters could be added, e.g., for contrast enhancing, smoothing, mixing, and so on. (Even the simplest 3×3 matrix filters can do amazing things.) It is not yet clear, however, what kinds of filters will turn out useful. Eventually, some fairly general filter *definition* formats ought to be given, enabling the incorporation of additional filters ad libitum.]

(For another type of filtering, cf. `F` below.)

2.14. *Comparing*

`.F03`

Copies the Yard into the Backyard.

`.F04`

Produces the pixelwise `xor` of the Yard, the Backyard and white (`:F`), and puts it into the Backyard. (Notice that doing it a second time reverts the effect.)

`.F05`

Like `.F03` but in the opposite direction.

`.F06`

Like `.F04` but in the opposite direction.

`.F07`

Exchanges the Yard and the Backyard. (*Exercise*: How can this be done *without* using `.F07` ?)

The most refined operation is

`F` f, o

executed between the Yard and the Backyard, the result going to the Backyard.

f and o may be given in any order, either or both may be missing.

f is the 'filter', specifying the 'colour components' of the pixels in the Yard that participate in the operation:

missing or `:0` or `:F` – all components;
`:1`–`:F` – the components for which the bit positions set in f are set in the colour number.

o is the operation between the 'permitted' components of the Yard and the (unfiltered) components of the Backyard:

missing – plain copying (the Backyard components play no role);
`~` or `N` – the complement gets copied (Backyard has no role);
`&` or `A` – the `and` of the two;
`|` or `O` – the `or` of the two;
`^` or `X` – the `xor` of the two;
`@` or `C` – the `xor` of the two, `xor`'ed with white.

When using the *letter* symbols, separate them from `F` by a space.

(So `F` in itself is equivalent to `.F03`, `F @` is equivalent to `.F04`. `F ^` is also reversible.)

[Maybe "white" ought to be altered to `g`.]

2.15. *Saving / Restoring*

`.PGU`

Saves the current contents of the Yard into free memory (if there is enough to hold it). Bit 9 in `@z` is set if successful, cleared if not (no space).

`.PGD`

Restores the Yard from what has been saved in memory.

Saving / restoring is "stacked": the picture saved last gets restored first.

(Saving the Yard into a *file*, cf. next sect., has no such restrictions but is somewhat slower and more circumstantial.)

2.16. *Picture Files*

> *file*

Puts the picture from the Yard into *file*: creates the file if it does not exist, overwrites it if it exists. If no extension is given, `.pic` is appended. *file* can be a text variable. Bit 10 in `@z` is set if successful, cleared if not.

< *file*

Puts the picture from *file* (produced by > *file* or by some extraneous process, e.g., a digitizer program) into the Yard. If no extension is given, `.pic` is assumed. *file* can be a variable. The file name is shown in the top line of the screen. `@z` as above.

< in itself simply cancels the reference to a previous picture. (`@z` unchanged.)

The physical format of picture files is as follows (no need to know).

They have a 64-byte *header*:

`:00–:07 SG` – signature,
`:08–:1F NN` – anything,
`:20–:2F DL` – directory link,
`:30–:33 SZ` – size in bytes, counted from byte `:30`,
`:34–:37 CD` – coding method,
`:38–:3B CP` – compression method,
`:3C–:3D WD` – picture width in pixels,
`:3E–:3F HT` – picture height in pixels.

In multibyte numbers, higher bytes precede lower ones (in contrast to the PC).

Settings for the present format are:

`SG` – `PICTURE#`,
`NN` – zero (may contain information for / by extraneous handling programs),
`DL` – zero (reserved for more complex file formats to come),
`SZ` – as it happens to turn out,
`CD` – `:012220C4`,
`CP` – `:00000044`,
`WD` – 296,
`HT` – 376.

From `:40` till the end, the *body* is encoded:

With *this* `CD` and `CP`, pixels are packed, left to right in a row and rows from bottom to top, as follows.

In each byte, bits 7–4 contain the colour number, bits 3–0 contain the repetition factor minus one. Rows are not straddled by counts (a new row starts a new byte).

[Complications may be added: Loading / storing of clippings (rather than of whole pictures), picture transformations *while* loading, etc.]

2.17. *Request Files*

`[` *file*

Loads a request file and executes it. If no extension is given, `.prq` is assumed. *file* can be a text variable. Bit 11 in `@z` is set if successful, cleared if not.

Another `[` *file* during request file execution kills (discontinues and closes) that file and starts the new one. A `[` in itself merely kills, and it does not affect `@z`.

(Notice that this arrangement enables 'chaining' of files: they may start one another, but there is no 'return'; 'back-call' is possible of course. [A 'call / return' discipline is implementable.])

File execution survives a `.DEL`, and a file itself may issue `.DEL` (not forgetting that it clears all stacks and variables).

Request files can be written on any editor capable of producing a 'character file' (and not putting in extra printing etc. instructions); 'verbose' coding is used.

They can contain any requests / terminators listed above (except `.BSP` and `.ESC` make no sense and act as `.ETR`).

In addition, the following 'program control' constructs are available.

`.CYC` / `.LVC`

Starts / ends a cycle (loop). Cycles can be nested (to a depth of 32); exceeding the nesting depth results in termination of file execution and closing the file; bit 31 in `@z` gets set. Multiple `.LVC`'s are permitted: each *executed* `.LVC` exits a cycle level. A supernumerary `.LVC` acts as an `.ETR` with a beep. (The cycle stack is maintained separately from the number stacks; so cycle structure and number stack dynamics are not bound to respect each other.)

`.RPT`

Jumps to the request line immediately following the topmost `.CYC` (the one on top of the cycle stack, that is, the `.CYC` of the innermost cycle it is in). A cycle may contain any number of `.RPT`'s. An `.RPT` that is not inside a cycle jumps to the beginning of the request file.

label:

(No spaces before the :.) Marks a target location to 'go to'; *label* is any name. Multiple use of the same label name is not inhibited (but is signalled); later occurrences override former ones.

^ *ll*

Goes to *ll*, *ll* being a label name or a text variable containing a label name.—Can also be used from console.

`.FIN`

Kills the file. End-of-file counts as `.FIN`. Bit 31 in `@z` gets set.

Input from file and from console can be intermixed arbitrarily:

Any keystroke coming at any time from the console

Takes over control from the file at the end of the current request line, and starts (or is) a console request. Mouse buttons (do not take control but) *halt* file execution as long as held down (mouse position does not change).

`.BRK` (from the file)

Hands over control to the console. Bit 30 in `@z` gets set.

`.TAB` (from the console)

Causes the file to resume control.

`.sTAB`

Another way to kill the file (to be used from console or a routine). Bit 31 in `@z` is set.

`.ECX`

Echo switch: At starting a request file, requests coming from it do not appear in the request line, nor do program responses in the response line. `.ECX` changes to echo mode (this might make sense for checking purposes), a second `.ECX` revokes it. (Console input is always echoed.) *Messages* are shown in either mode.

`.sF01`–`.sF03` in mode 12–13 (prompt for picture lettering) prompts the person at the console. (For lettering from the request file, use _ or | .) If terminated by `.TAB`, the file regains control; otherwise, control remains with the console.

The request file is executed from disc, merely label / cycle information is hashed / stacked in memory. The file gets closed at execution completion.

*

A request file named `pictdemo.prq` may have been placed in the DOS directory *where* `picture` is being run. If such a file is present, and no library is attached to the program, `.INS` starts that file. (If a library is attached, `.INS` acts differently, cf. below; it is for the library to provide demonstrations or not.)

2.18. *Libraries*

{ *file* or { *file*1 , *file*2

Builds a (single) library from the library file(s). If no extension is given, `.prl` is assumed.

The file(s) can also be given at program start in the parameter list to `picture` (without the {).

Another { with argument(s) drops the library and builds the new one. A { in itself merely drops it.

Library attachment survives a `.DEL`; a routine itself may issue `.DEL`, but argument, cycle and stack structures (if any) will break down (so do not do it from some obscure recess of an intricate Gaudi-style edifice, in lack of a safe escape).

Library files can be written the same way as request files.

A library consists of routines (routine definitions). It may contain up to some 6000 names, otherwise its size is not restricted (unless by memory space).

Routine call is the routine name, followed by its arguments (if any); the format is the same as with ordinary request sentences.—Nine routines may get (in addition to calling them by name) one-keystroke activators: `.aF01`—`.aF09` can be defined to call a routine. (`.aF10` is reserved for esoteric purposes.) This is mostly intended for argumentless routines; if there *are* arguments, they are considered 0, except simultaneously pressing `s` sets the first argument, `c` the second argument to 1. Activators act also as terminators.

A routine can be defined to have up to 9 arguments, possibly none. Routine arguments can be number constants and variables and text variables (not text constants and not terminators), and cannot be used in place of the operation code (e.g., an argument being a text variable containing `ifeq` cannot be used as `ifeq`).

(Routines can of course use values in any way, without making them formally into 'arguments'.)

Routine bodies may contain any request sentences / lines, including routine calls.

The following constructs are available (in addition to those already mentioned).

routine|*n*:

(No spaces around the | and before the :.) Routine definitor, to be followed by the routine body; n is the number of routine arguments ($0 \leq n \leq 9$). *routine* must be a name different from the names used for 'basic' requests (e.g., `ifeq` is not accepted as a routine name). The body ends at the next routine definitor or at the end of the file.

n: *routine*

(No spaces before the :.) `.aF0`n ($1 \leq n \leq 9$), with `s`, `c` possibly also depressed, is to activate *routine*.

#*m*

(No spaces after the #.) Reference, in the routine body, to routine arguments, $1 \leq m \leq n$. (A # within a text string is not considered an argument reference.)

.RET

Return from routine. Body end implies a .RET even without saying so.

.XIT

Exits, meaning return not only from the current routine but also from all its direct and indirect callers.

label: and ^ *ll*

As with request files, but pertains here to the library (no relation to possible identical names elsewhere, not accessible to jumps from outside and unable to jump outwards). Label names may conflict with everything except routine names; a routine name is a legitimate target of a 'go to': jump to the routine start. (Notice that a label seemingly *preceding* a routine definitor belongs, according to the above rules, to the body of the previous routine. A label may of course immediately *follow* the definitor.)

.CYC / .LVC / .RPT

This, too, is local to the library; a separate stack is maintained for them (also 32 deep). (Cycles in routines and cycles in request files have no relation to each other.) An .RPT without a corresponding .CYC is not defined here.

.BRK (from a routine)

Re-calls the caller (console or request file, as the case is); clears bits 31–16 (the two highest bytes) in @z. (Bits 23–12 are never set and bits 15–12 never cleared by the program, nor will be by later extensions; they can be used at the customer's discretion.)

.INS (from the console or a request file)

Returns from a re-call to the re-calling routine; sets bit 29 in @z. (When no library is attached, .INS starts a demo file.)

A .BRK from a request file, if given in a re-call, also returns to the routine. (So a request file needs no knowledge of who it is communicating with, whereas the routine can know it.)

When a routine prompted the console for picture lettering, both .INS and .ETR return to the routine.

.sINS (from the console or a request file)

Exits (causes to .XIT) the current routine.

{

Given from within a routine (with or without arguments) exits everything before executing.

`.cF02`

Shows library names and activators in the Backyard.—Saves and restores the Backyard (if unable, asks the console, whether to proceed).

Multiple use of the same name is not inhibited (but is signalled); later occurrences override former ones.

At call, the number of arguments need not conform with the definition: they are taken in order, and if less, missing arguments are considered 0, if more, supernumerary arguments are simply neglected.

Routines may call each other and themselves to (practically) any depth. (The "official" stack is used; if the depth happens to be exhausted, increase *that*—yet the probable cause is an endless recursion.)

Re-call is confined to one level. (So a routine called from within a re-call cannot itself re-call again.) Prompt for picture lettering, as well as asking for a keystroke, does not count as re-call.

No request file can be started from a re-call. (But of course the routine can be induced to start the file after the re-call, or the file can be started beforehand.)

Go to's and cycles are permitted to hop around between *different* routines. (This, however, is a risky business.)

All errors in routine definition and usage prompt the console to choose between 'skip' and 'exit'.

The library is built and stored within memory; the defining file(s) will be closed immediately.

(The functional picture behind this is as follows. A customer might want access to the tested-and-proved, time-honoured routines of the application, and, at the same time, routines for his / her / its favourite keystroke sequences, for one-time arrangements, and for experimentation; the 'personal' routines possibly partly replacing / overriding the 'institutional' ones.—Anyhow, applications ought to have conventions as to access procedures, name usage, saving or not of variables, etc., etc.)

[Consider the following variants.

Using a programmed stack, rather than the official one, for routine nesting. (Unfortunately, DOS does not behave intelligently at stack overflow.)

Multi-level re-call.

Label range local to routines.

Strong argument checking (less comfortable but more foolproof).

Removing the restrictions on arguments.

Smaller limit on the number of arguments (say 3 instead of 9—slightly reducing the processing costs).]

Appendix. Process Interleaving

With programs like this, it is often desirable to be able to run other processes in parallel: make a hard-copy, search in alien databases, run statistics, and the like.

A PC is not a real multiprocessor (not under DOS, not under Windows). But it can be made look "as if".

In addition to more complicated solutions, the program offers a very simple way to bind it into any bunch of concurrent processes: Its executive routine, `exec`, executes at every call a single request line, then returns. So, instead of running the program itself, its `exec` can be run from anywhere, interleaved with anything.

The exact details are as follows.

The only file to be `#include`'d is `pictools.h` (it brings the rest).

The `main` of the stand-alone program is

```
if ( init(argc,argv) ) return(1);
while ( exec() ) ;
finish();
return(0);
```

Now, rather than running it as it is, call its components from whatever controls the program flow:

Once `init` has been (successfully) executed, `exec` can be activated at wish; it returns 0 for `.END`, 1 for everything else. When enough of it, `finish`. (Re-`init` after `finish` is permitted.)

`init` goes to graphics mode and grabs the screen; `finish` releases graphics. Concurrent processes may access the screen if they know what they do (and even if they don't); they may save-and-restore it or leave visible traces or drop the current graphics mode altogether.

To supply an additional or alternate input source, say, `newin`, execute at any time `extrin = newin`. If no longer needed, drop it by `extrin = dummy`. `newin` has to be a function with one `int` argument, returning an `int`. The argument value is of no interest. (It is zero when called before request line start, non-zero afterwards.) The function shall return zero if willing to give precedence to other input sources; it will be called again later. When wanting to have a go, but having nothing to send, it shall return `:FF`, which maintains rights and elicits repeated calls. When wanting to send, it shall return `:FE`, which will be dropped, and start issuing 'veritable' (non-zero and non-`:FE`–`:FF`) characters, these will be taken at face value until line termination. The line *must* terminate. (If zero is to be given, which is never the case, give `:7F` instead, which is anyhow what the outcome is.) For the next line, `newin` will be called first, so

it may maintain precedence and continuity. Both concise and (hence) verbose coding may be used.

No control-transfers (go to's, cycles) are feasible in `newin`. (Routine calls and re-calls are possible of course, even depending on conditions.) `.BSP`, `.ESC` act as `.ETR`.

`.DEL` is survived by `newin`.

[An added degree of freedom could be: Make `exec` call arbitrary extraneous functions, periodically or at request. Do we need that?]

Request Summary

(Visible screen symbols are not listed; they have been felt self-explanatory. Mouse actions are too multifarious to be summarized here; remember this much: left paints, middle stains or marks, right erases or drops.)

Bracketed numbers refer to subsections of Sect. 2.

; – request separator [0] ␣ or , – argument separator [0]
.ETR – request line terminator [0]
.BSP, .ESC – editing keys [0]
.END – program end [1]
.DEL – program restart [1]
.F01 – clear Yard [8]
.F02 – clear Backyard [8]
.F03 – Yard → Backyard [14]
.F04 – Yard xor Backyard xor white → Backyard [14]
.F05 – Backyard → Yard [14]
.F06 – Yard xor Backyard xor white → Yard [14]
.F07 – Yard ↔ Backyard [14]
.F08 – contour filter, four neighbours [13]
.F09 – contour filter, eight neighbours [13]
.F10 – complement the palette [5]
.sF01 – = left click [4]
.sF02 – = middle click [4]
.sF03 – = right click [4]
.sF04 – wait for keystroke, set @x [3]
.cF01 – help [1]
.cF02 – library contents [18]
.cF03 – display current palette [5]
.cF04 – display current font [10]
.cF05 – count pixels of current colour in the Yard, set @x [5]
.cF06 – stack load, set @x, @y [2]
.cF07 – date and time, set @x, @y [1]
.cF08 – convert date, set @y [1]
.cF09 – inquire free memory, set @x, @y [1]
.aF01–.aF09 [+ s, c] – routine activators [18]
.RAR / .LAR – drawing colour selection [5]
.UAR / .DAR – action mode selection [7]
.PGU – save Yard (to memory), set @z [15]
.PGD – restore Yard (from memory) [15]

.sPGU – mouse symbol switch [4]
.sPGD – mouse visibility switch [4]
.INS (if no library) – show demo [17]
.INS (with library) – return from re-call, set @z [18]
.sINS – escape from routine [18]
.TAB – hand back control to request file [17]
.sTAB – abandon request file, set @z [17]
.BRK – hand over control to / re-call console, set @z [17,18]
.FIN – end of request file, set @z [17]
.ECX – echo switch [17]
.CYC – cycle start [17,18]
.LVC – cycle end [17,18]
.RPT – cycle repeat [17,18]
.RET – return [18]
.XIT – exit [18]
= x, y – mouse positioning [4]
= (without arguments) – inquire mouse position / pixel colour, set @x, @y, @z [4]
SPEED n – mouse speed [4]
SPEED (without argument) – inquire mouse speed [4]
S l – line type [6]
\ pal, col, r, g, b – palette colour definition [5]
/ pal – palette [5]
N $col\,1, col\,2, col\,3$ – notification colours [5]
G col – background colour [5]
H col – halting colour [5]
C col – drawing colour [5]
X $col\,1, col\,2$ – change $col\,1$ to $col\,2$ in the Yard [5]
Y $col\,1, col\,2$ – mix $col\,1$ with $col\,2$ in the Yard [5]
Z col – blink col on the screen [5]
P – point [8]
Q – point, marked [8]
L dx, dy – straight line segment [8]
K dx, dy – straight line segment, marked [8]
A r, f, t – circular arc [8]
E r, f, t, d – elliptical arc [8]
B bx, by, cx, cy, dx, dy – Bernstein arc [8]
O i – override the interval number for the next Bernstein arc [8]
M $mode$ – action mode [7]
* f – font [10]
_ '$text$ – horizontal lettering [10]
| '$text$ – vertical lettering [10]
U m – magnification factor [12]
F f, o – filter-and-mix [14]

> *file* – store Yard to *file*, set @z [16]
< *file* – load Yard from *file*, set @z [16]
< (without argument) – drop current file [16]
[*file* – execute request file, set @z [17]
[(without argument) – drop current request file [17]
{ *file* or { *file*1 , *file*2 – build library [18]
{ (without argument) – drop library [18]
any keystroke – take over control by console [17]
label: – label [17,18]
^*ll* – go to *ll* (constant or variable) [17,18]
routine [*arguments*] – routine call [18]
routine|*n*: – routine definitor [18]
n: *routine* – routine activator definition [18]
#*n* – argument reference [18]
cond *x*, *y* (cond = IFEQ, IFNE, IFGT, IFLE, IFSET, IFCLR) – conditionals [3]
@a–@z, @A–@Z – number variables / references [2]
$*n*, $^*n* / &*n*, &^*n* – plain / library stack references [2]
+ *x* or ↑ *x* – push number constant or variable, set @z [2]
- *x* or ↓ *x* – pop to number variable [2]
~ *x* – alter stack pointer [2]
=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, ? – arithmetic operations [2]
n – shorthand for @x = *n* (*n* a constant) [2]
n, *m* – shorthand for @x = *n*; @y = *m* (*n* and *m* constants) [2]
@0–@9 – text variables [2]
=, ~, ? – text operations, ~ sets @z, ? sets @x and @y [2]
'*u* – shorthand for @9 = '*u* [2]
"*u* – message [2]
SYS *command* – call DOS [1]
SYST *command* – call DOS, restore everything [1]
SYSTEM *command* – call DOS, restore and wait [1]
ERR *c* – errors to be notified [1]
% *anything* – comment [1]