PC Scheme–Geneva:
User's documentation

Larry Bartholdi Marc Vuilleumier

December 22, 1993

Abstract

"My English is worse than the garden of my uncle"
—Unknown Chinese philosopher, about 4000 years ago

This document describes various major extensions to PcScheme by the Geneva Scheme Team. These include BGI, a graphics package based on the interface developed by Borland, and Mouse support. More documentation about Ed will be added here as soon as available.

BGI is distributed by Borland as C and Pascal libraries. Our adaptation to Scheme preserves the spirit of Borland's original product.

Other documentat about PC Scheme/Geneva are: **SCHEME.TEX**, the release log file. You should read it, as it contains the most recent informations about PC Scheme/Geneva; **REFCARD.TEX**, our Quick Reference Card. It contains all about PC Scheme/Geneva, including default key sequences for Edwin and Ed.

The authors' work is supported by the University of Geneva. They can be reached at their E-Mail address "schemege@cui.unige.ch". This document was typeset by LaTeX.

Contents

Part I

Ed: a Generic Editor

ED is an object-oriented editor toolkit using a subset of BRIEF's key sequences; you can drive it from your programs or simply use it as a text editor. You can easily remap any key sequence to any function, or even implement new features. Basically, ED provides four levels of editor that you access through four creators (meta-functions):

`(make-editor [window] ['EXIT-FREELY])`	$\Longrightarrow$	*a basic editor object*
`(make-enhanced-editor ...)`	$\Longrightarrow$	*ditto, with additional features*
`(make-color-editor ...)`	$\Longrightarrow$	*ditto, with block highlight*
`(make-scheme-editor ...)`	$\Longrightarrow$	*ditto, with scheme-specific rules*

Typical use:

```
(define ed (make-[...-]editor))        ; create a full-screen editor
(ed 'REMAP-KEY 27 '@GOTO-LINE)         ; remap Esc to "go to line"
(ed [filename])                        ; open the editor
```

If you want a multi-window editor, you can define two or more editors in different window ports; scrapbook and keystroke assignements will be shared by all.

This part has been written by Marc Vuilleumier.

Chapter 1

Keystrokes

1.1 Basic Editor Commands

Step		$\boxed{\uparrow}$		**Jump**	PgUp	
	$\boxed{\leftarrow}$	✥	$\boxed{\rightarrow}$	Ctrl-$\boxed{\leftarrow}$	✥	Ctrl-$\boxed{\rightarrow}$
		$\boxed{\downarrow}$			PgDn	
∞		Ctrl-PgUp		**Delete**		
	Home	✥	End	BackSpace	✥	Del
		Ctrl-PgDn				

Alt-I	toggle insert/overwrite mode
Alt-G	go to line
Alt-K	kill to end of line
Alt-D	delete current line
Ctrl-L	refresh display
F7	record a keystroke sequence
F8	replay a keystroke sequence
F10	execute a command by name
Alt-E	edit another file
Alt-W	write file to disk
Alt-O	choose new output name
Alt-R	read a file into current
Alt-X	exit editor

1.2 Enhanced Editor Commands

Use a Color Editor if you want block highlight features.

Alt-M	mark block
Alt-L	line block
Alt-C	column block
⟨Keypad −⟩	cut block/line to scrap
⟨Keypad +⟩	copy block/line to scrap
⟨Keypad /⟩	invert block bounds (swap anchor)
Ins	insert scrap into text
Del	delete block
Alt-W	write block/file to disk
Alt-[1 . . . 3]	drop a bookmark
Alt-J [1 . . . 3]	jump to bookmark
Ctrl-R	replicate a command
Ctrl-F5	toggle case sensivity
Alt-S or F5	search string
Shift-F5	repeat previous search
Alt-T or F6	translate string
Shift-F6	repeat previous translate

1.3 Scheme Editor Commands

Autoindent is active and color changes on each parenthesis level.

Ctrl-A	enlarge mark around Scheme expression
Ctrl-Z	mark the biggest Scheme expression
Ctrl-F10	evaluate marked expression
Alt-F10	evaluate current file
Tab	complete symbol or reindent line/block
Shift-Tab	prepare to add a comment
Alt-Q ⟨*key*⟩	use *key*'s basic definition

Chapter 2

Driving an Editor

`(ed [message] [arguments])`	*message* defaults to *'OPEN*
'OPEN [*filename*]	pop-up the editor, handle commands
'SAFE	ensure current buffer is saved
'CLEAR	clear buffer
'NAME [new-value] .... ⇒	name of buffer
'BUFFER [new-value] .... ⇒	a list of strings
'POSITION [new-value] .... ⇒	cursor position
'INSERT [new-value] .... ⇒	`#F` when in overwrite mode
'TAB [new-value] .... ⇒	tab expansion width
'TABULIZE-MODE [new-value] .... ⇒	*'COMPRESS*, *'NORMAL* or *'EXPAND*
'SEPARATORS [new-value] .... ⇒	string (used for word move)
'CASE-SENSIVITY [new-value] .... ⇒	`#F` when disabled
'COLORS [new-value] .... ⇒	associative color list
'COMMENT-COLUMN [new-value] .... ⇒	column #
'INDENT-TOKENS [new-value] .... ⇒	list of special tokens
'INPUT-PORT [new-value] .... ⇒	port used for input
'DO-STRING *string*	feed editor with keystrokes
'READ-ACTION .... ⇒	an *action*: the next event
'HANDLE-ACTION *action*	process one command
'REMAP-KEY *key function*	

key is `(list* [...` ASCII$_2$`]` ASCII$_1$`)`
function if `(list* [(`*context action*`) ...]` *default-action*`)`
context is $\lambda(ed) \longrightarrow$ boolean
action is a character, a string, or a symbol:

'@LEFT	*'@WORD-LEFT*	*'@HOME*	*'@RIGHT*
'@WORD-RIGHT	*'@END*	*'@UP*	*'@PAGE-UP*
'@TOP-OF-BUFFER	*'@DOWN*	*'@PAGE-DOWN*	*'@END-OF-BUFFER*
'@DEL	*'@DELETE-TO-EOL*	*'@DELETE-LINE*	*'@BACKSPACE*
'@TAB	*'@ENTER*	*'@QUOTE*	
'@INSERT-MODE	*'@GOTO-LINE*	*'@REFRESH*	*'@RECORD*
'@PLAY	*'@EXECUTE*	*'@LOAD*	*'@READ-INTO*
'@WRITE	*'@RENAME*	*'@EXIT*	
'@SEARCH	*'@REPEAT-SEARCH*	*'@CASE-SENSITIVITY*	*'@TRANSLATE*
'@REPEAT-TRANSLATE	*'@REPLICATE*	*'@BOOKMARK-*[1...3]	*'@JUMP-TO-*[1...3]
'@MARK-BLOCK	*'@LINE-BLOCK*	*'@COLUMN-BLOCK*	*'@CUT-BLOCK*
'@COPY-BLOCK	*'@INSERT-BLOCK*	*'@CANCEL-BLOCK*	*'@SWAP-ANCHOR*
'@MARK-EXPR	*'@MARK-DEF*	*'@SCHEME-PARENTHESIS*	*'@COMPLETION*
'@INDENT	*'@COMMENT*	*'@EVAL*	*'@EVAL-BLOCK*
'@SCHEME-ENTER			

Part II

Mouse Support in PcScheme

Microsoft(TM) specified a standard interface to drive the mouse from within applications. Practically all mouse vendors adhered to this interface, so it made sense to access the mouse though it from within PcScheme. *Unfortunately, the standard was designed for interfacing to assembly language and thus contains some peculiarities, like event handlers, that have carefully to be dealt with.*

The solution proposed is a slight extension of PcScheme*'s core—the virtual machine. An op-code has been added to support the* `int 33h` *calls, as well as an assembly-language event handler that dispatches requests to a Scheme closure. The solution is not perfect in the sense event handling cannot be honored in the middle of a basic instruction (for instance, during a call to* `readline`*). The slowdown due to Scheme handling of even the most simple events (like recording a move), and the context switching required, may render event handling unpractical on slow processors; the system has nevertheless successfully been used on a fast 80486 PC.*

This part has been written by Larry Bartholdi.

Chapter 3

The interfacing

The most important differences between the assembly level interface and the Scheme procedures are:

- PcScheme has an Object-Oriented interface
- PcScheme uses symbolic constants abundantly (and list of constants instead of bit masks)

On the lowest level, all requests are executed through

(%MOUSE *ax bx cx dx si di* [*es:dx*])

which returns a list: (*ax bx cx dx*) of the CPU registers after the call has been effected. Fortunately you will never have to call this function directly; but this is how PcScheme's interface works.

The high-level interface (in `MOUSE.FSL`) works with a single "mouse" object,

(MOUSE *message* [*parameters*]) $\hookleftarrow$

that translates the message to `(%mouse)` calls. The messages `mouse` recognises are:

Message Args ⇒ Result	**Comments**
'RESET ⇒ *#-buttons*	This message should be the first sent when dealing with the mouse. It returns -1 if no mouse is avaliable.
'SHOW	Increments the mouse's visibility count. The count initially is -1, can be incremented or decremented, but can never become positive (calls to *'SHOW* are without effect if the count is ≥ 0). The pointer is visible only when the count is $= 0$.
'HIDE	Decrements the mouse's visiblity count.
'INQ ⇒ *(buttons-down x y)*	*buttons-down* is a list containing any of *'LEFT*, *'RIGHT* and *CENTER*. The x– and y-positions are thought in 640×200 units, although they can be redefined to any scaling. They are always in linear correspondence with the "Mickeys".

Message Args ⇒ Result	Comments
'MOVE x y	Moves the pointer to an absolute location. As always, $(0, 0)$ is the upper left corner. The values are clipped to the limiting rectangle.
'PRESS *button* ⇒ *(buttons-down count x y)*	There is an additional field compared to *'INQ*: the number of times the button has been pressed. All other values refer to the mouse state when the last "press" occured. The count is reset to 0 after this call. *button* is a symbol.
'RELEASE *button* ⇒ *(buttons-down count x y)*	Just like *'PRESS*, except it returns the release count.
'LIMITS 'HORIZONTAL x_0 x_1	Sets the mouse's horizontal viewport. It will not be able to move out of that area. If it is outside at the moment of the call, it will be projected to the border.
'LIMITS 'VERTICAL y_0 y_1	Sets the mouse's vertical viewport.
'LIMITS 'BOTH x_0 x_1 y_0 y_1	Sets the mouse's viewport.
'SHAPE *shape*	Sets the graphics pointer. *shape* is a list: (*x-hot-spot y-hot-spot and-mask xor-mask*), where the masks are lists of 16 integers. This function does not affect the text-mode pointer.
'CURSOR *name* ⇒ *cursor*	Returns a predefined pointer for *'SHAPE*. Two sources were used to grab bitmapped pointers: the X Windows shared library (in `openwin/share/lib/include/bitmaps`), and an MsDos package called "Precise Point". Both sources were used without permission. *name* is a symbol, and can have values *'CENTER*, *'KEYBOARD*, *'LEFT*, *'RIGHT*, *'STAR6*, *'TARGET* (X Windows) and *'ARROW*, *'BLOCK*, *'CIRCLE*, *'EXCLAIM*, *'HAND*, *'HOURGLAS*, *'KITE*, *'MESH*, *'SMALL*, *'SQUARE*, *'STAR4*, *'TEXT*, *'X* (Precise Point).

'TEXT-TYPE 'SOFTWARE *and* *xor*	Sets the text-mode pointer to software, i.e. to modification of characters on the screen. *and* and *xor* are pairs: (*character . attribute*). As a simple example,
	`(mouse 'TEXT-TYPE 'SOFTWARE`
	`        (cons (integer->char \#xff) \#xff)`
	`        (cons (integer->char 0) \#x80))}`
	would make the character under the pointer blink.

'TEXT-TYPE 'HARDWARE *start* *end*	Sets the text-mode pointer to hardware, i.e. uses the cursor generator of the graphics adapter. *start* and *end* are the start– and end-rows of the cursor, usually in range 0–8 or 0–16. Note that the cursor then serves two uses; this mode should be useful especially in an editor, if the mouse moves are interpreted as cursor positioning.
'MICKEYS ⇒ *(x y)*	Returns the mouse's movement in Mickeys since the previous call to this function. A Mickey is the mouse's fundamental unit, ≈ 1/200inch.

Message Args ⇒ Result	Comments
'HANDLER handler ⇒ *old-handler*	Installs an event handler. A handler is a list: (*events . procedure*), where *events* is a list containing any of *'MOVE*, *'LEFT-DOWN*, *'LEFT-UP*, *'LEFT* (both left-button messages), *'RIGHT-DOWN*, *'RIGHT-UP*, *'RIGHT*, *'CENTER-DOWN*, *'CENTER-UP*, *'CENTER*, *'DOWN*, *'UP*, and *'BUTTONS*. When one of these events occurs, *procedure* will be called. *procedure*'s arguments are (*triggering-events buttons-down x y mickeys-x mickeys-y event-time*). Note that *triggering-events* is a list, because it may contain more than one event. An important thing to mention is that normally *procedure* will not be reinvoked before it returns. Events that occur during *procedure*'s execution are lost. In particular, if *procedure* does not return (in case of error, for instance), mouse events remain disabled. They can be re-enabled with the next command.
'ENABLE	Re-enables event handling. With this, re-entrant mouse-handlers can be written! In its normal operating mode, `mouse` stacks up mouse messages and dispatches them one at a time, sending the next message when the dispatch of the previous one returned. If this function is called, the message-dispatching system is spawned; in effect, calling this function from within an event handler may result in the handler being called "re-entrantly".
'DISABLE	This message suppresses re-entrancy by cancelling the effect of a previous *'ENABLE*.
'PEN-ON	Microsoft Manual says "**Light-Pen emulation:** Enables light pen emulation by the mouse driver for IBM BASIC. A "pen down" condition is created by simultaneously pressing the left and right mouse buttons." I tried darn hard to understand this, without success.
'PEN-OFF	ditto.
'MICKEY-RATIO x y	Sets the number of Mickeys per 8 pixels for horizontal and vertical mouse motion.
'EXCLUDE x_0 x_1 y_0 y_1	Defines a rectangle within which the mouse is invisible.
'SPEED-THRESHOLD speed	Sets the speed limit between simple– and double-speed motion, in Mickeys per second.
'SENSITIVITY [x y speed] ⇒ *(x y speed)*	Reads or sets the Mickey ratio and speed threshold.
'INTERRUPT-RATE rate	*rate* can be *'NONE*, 30, 50, 100 or 200, and governs the tradeoff between graphic resolution and application performance. It is applicable only to the InPort mouse (see *'INFORMATION*).
'POINTER-PAGE [page] ⇒ *page*	Reads or sets the display page for the mouse pointer.
'LANGUAGE [language] ⇒ *language*	Reads or sets the language for mouse driver messages. *language* can be any of *'ENGLISH*, *'FRENCH*, *'DUTCH*, *'GERMAN*, *'SWEDISH*, *'FINNISH*, *'SPANISH*, *'PORTUGESE* or *'ITALIAN*.

Message Args ⇒ Result	Comments
`'INFORMATION` ⇒ *(version type IRQ)*	Returns information on the mouse driver. *version* is a floating-point number. *type* is a symbol: `'BUS`, `'SERIAL`, `'INPORT`, `'PS/2` or `'HP`. *IRQ* is an integer.

Chapter 4

A simple example

I include here a simple demo of what can be done with graphics and mouse: a program that draws lines at random while the pointer (representing two eyes) winks at the user when he presses the buttons.

```
;* EYES.S

;************************************************************************

;* *

;* PC Scheme/Geneva 4.00 Scheme code *

;* *

;* (c) 1985-1988 by Texas Instruments, Inc. See COPYRIGHT.TXT *

;* (c) 1992 by L. Bartholdi & M. Vuilleumier, University of Geneva *

;* *

;*----------------------------------------------------------------------*

;* *

;* A Simple Mouse Demo *

;* *

;*----------------------------------------------------------------------*

;* *

;* Created by: L. Bartholdi Date: 19930930 *

;* Revision history: *

;* - 18 Jun 92: Renaissance (Borland Compilers, ...) *

;* *

;* ``In nomine omnipotentii dei'' *
```

```
;**********************************************************************

; 0 . . . . . . . . . . . . . . . .
; 1 . . . . . . . . . . . . . . . .
; 2 . . o o o . . . . . . . . . . .
; 3 . o . . . o . . . . . . . . . .
; 4 . o . . . o . . . . . . . . . .
; 5 o . . . . . o . . . . . . . . .
; 6 o . . . . . o . . . . . . . . .
; 7 o . . . . . o . . . . . . . . .
; 8 o . . . . . o . . . . . . . . .
; 9 o . o o o . o . . . . . . . . .
;10 o o . . . o o . . . . . . . . .
;11 . o . . . o . . . . . . . . . .
;12 . o . . . o . . . . . . . . . .
;13 . . o o o . . . . . . . . . . .
;14 . . . . . . . . . . . . . . . .
;15 . . . . . . . . . . . . . . . .
;  151413121110 9 8 7 6 5 4 3 2 1 0

(define open-eye '(0 0 ( #b0000000011111111
 #b0000000011110111
 #b0000000011100011
 #b0000000011000001
 #b0000000011000001
 #b0000000010000000
 #b0000000010000000
 #b0000000010000000
```

```
 #b0000000010000000
 #b0000000010000000
 #b0000000010000000
 #b0000000011000001
 #b0000000011000001
 #b0000000011100011
 #b0000000011110111
 #b0000000011111111 )

       ( #b0000000000000000
 #b0000000000000000
 #b0000000000011100
 #b0000000000100010
 #b0000000000100010
 #b0000000001000001
 #b0000000001000001
 #b0000000001000001
 #b0000000001000001
 #b0000000001011101
 #b0000000001100011
 #b0000000000100010
 #b0000000000100010
 #b0000000000011100
 #b0000000000000000
 #b0000000000000000 )
       ))
(define closed-eye `(0 0 ,(caddr open-eye)
         ( #b0000000000000000
```

```
   #b0000000000000000
   #b0000000000011100
   #b0000000000100010
   #b0000000000100010
   #b0000000001000001
   #b0000000001000001
   #b0000000001011101
   #b0000000001111111
   #b0000000001110111
   #b0000000001100011
   #b0000000000110110
   #b0000000000111110
   #b0000000000011100
   #b0000000000000000
   #b0000000000000000 )
       ))

(define (right pattern)
  (map (lambda (x) (* x #x100)) pattern))

(define (join p1 p2)
  (map (lambda (x y) (bitwise-or x y))
       p1 (right p2)))

(define m0 (list 0 0
 (join (caddr open-eye) (caddr open-eye))
 (join (cadddr open-eye) (cadddr open-eye))))
(define m1 (list 0 0
```

```
 (join (caddr open-eye) (caddr closed-eye))
 (join (cadddr open-eye) (cadddr closed-eye))))
(define m2 (list 0 0
 (join (caddr closed-eye) (caddr open-eye))
 (join (cadddr closed-eye) (cadddr open-eye))))
(define m3 (list 0 0
 (join (caddr closed-eye) (caddr closed-eye))
 (join (cadddr closed-eye) (cadddr closed-eye))))

(init-graph)
(mouse 'RESET)
(mouse 'SHOW)
(mouse 'SHAPE m0)
(mouse 'HANDLER `((LEFT RIGHT) .
  ,(lambda (event state . rest)
     (mouse 'SHAPE
    (cond
      ((equal? state '()) m0)
      ((equal? state '(LEFT)) m1)
      ((equal? state '(RIGHT)) m2)
      ((equal? state '(LEFT RIGHT)) m3))))))
(writeln "Press any key to abort...")
((rec loop
   (lambda (count)
     (when (not (char-ready?))
       (let ((fade (* 100 (exp (/ (* count count) -40000.0)))))
 (if (> (random 100) fade)
     (begin
```

```
       (mouse 'HIDE)
       (mouse 'SHOW)
       (set-color 0))
     (set-color (1+ (random (-1+ (get-max-color))))))
       (line (cons (random (car (get-max-xy))) (random (cdr (get-max-xy))))
     (cons (random (car (get-max-xy))) (random (cdr (get-max-xy)))))
       (loop (1+ count)))))
 0)
(read-char)
(close-graph)
(mouse 'RESET)
```

æ

Part III

BGI: The Graphics Package

Written by Marc Vuilleumier

Last change October 5, 1993. First draft November 27, 1992

Chapter 5

User's manual

5.1 What is BGI?

BGI is a powerful, standard, device-adaptable interface for graphic output. It was introduced by Borland compilers, and give programmers a complete set of graphic primitives, working on every device for which a BGI driver exist. Included with this version of PcScheme, you'll find drivers for Cga, Mcga, Att-400, Ega, Vga, Hercules, Ibm-8514, Pc-3270, Hp95lx and true compatibles.

You may use any other BGI driver found on public sites or provided by your graphic hardware vendor (such as the `SVGA.BGI` driver for Super-Vga designed for Borland C 3.0 to improve the resolution and color capabilities.

You may also use the two drivers one of us (lb) has written, that are distributed with PcScheme: `SPY.BGI` and `PS.BGI`. The former is a debugging aid that prints to the screen all the driver requests. The latter creates a PostScript document that can then be incorporated in a text processor or sent to a printer. See section 5.5 for more information.

5.2 What is different in PcScheme's implementation of BGI?

- The name of primitives and constants is "Scheme-like" splitted in words:

Borland C 3.0		PcScheme
`initgraph(...)`	⟹	`(init-graph ...)`
`setcolor(`*LIGHTBLUE*`)`	⟹	`(set-color '`*LIGHT-BLUE*`)`

- Since graphic constants are contextual (which is not the case in C), we have removed redundancy in symbol names:

Borland C 3.0		PcScheme
`setlinestyle(`*DOTTED_LINE*`, ...)`	⟹	`(set-line-style '`*DOTTED*` ...)`
`settextstyle(`*GOTHIC_FONT*`, `*HORIZ_DIR*`, 4)`	⟹	`(set-text-style '`*GOTHIC HORIZ*` 4)`

- Some parameters are optional. The following commands, for instance, are all equivalent:

```
(init-graph 'DETECT 0 "")
(init-graph 'DETECT 0)
(init-graph 'DETECT)
(init-graph)
```

- Static structures have been replaced by lists. For instance, a polygon is not an integer and an array, but simply a list of points.
- The coordinate system can be completely adapted by the user.
- Twin-functions returning X and Y values have been joined and produce a point (i.e. a pair). This is true for text measurement, too.
- Special memory allocation functions have not been included.

5.3 Coordinate systems

Most graphic primitives have to place objects, on the screen for instance, given some coordinates; we call a "point" the information necessary to explicit a location on the output device.

By default, a point is a pair of whole numbers, the X and Y coordinates, and therefore it is highly device-dependent: using a Color Graphic Adapter (CGA) mode, for instance, the valid coordinate range is *'(0 . 0)* through *'(319 . 199)* in 4-color mode, and *'(0 . 0)* and *'(639 . 199)* in Black&White mode. With a Hercules-monochrome adapter, the range becomes *'(0 . 0)* through *'(719 . 347)*, and so on; *'(0 . 0)* is always the upper left corner of the screen, contrary to mathematical tradition. Note that BGI routines usually clip their parameters to the screen rectangle.

Now PcScheme allows you to freely define the coordinate system you'd like to use. You only need to specify the desired coordinates of the upper left and bottom right corner, no matter which video adapter or other device is present. A typical sequence would be:

```
(init-graph)                                ; detect and initialize video adapter
(set-world!  '(-100 . -100) '(100 . 100))   ; set the screen coordinates with origin centered
(put-pixel '(0 . 0) 'BLUE)                  ; plot a blue pixel at center of screen
```

Note that the X and Y coordinates are now real numbers, so you are allowed to call `SET-WORLD!` with parameters like:

```
(set-world!  `((- ,pi) . 1) `(,pi . -1)) ; ideal to plot a sine...
```

Of course, if you need to scan every pixel, you can use `GET-MAX-XY` to obtain the actual resolution of your screen adapter.

If you are urging to try BGI by yourself, skip to part 6, page 24. What follows is an extension to the BGI standard.

5.4 And further...

For type checking of point parameters, you are allowed to use the `SET-POINT?-!` primitive to change the point type. For instance, the following code would allow points to be a pair of real (this is done automatically when you use `SET-WORLD!`).

```
(define (real-point point) ; Type-checking for points
  (and (pair? point)
       (number? (car point))
       (number? (cdr point))))

(set-point?-! real-point)  ; Use the real-coordinates system
```

Now imagine you want to draw a bar chart using an exponential scale. Instead of adding a call to the LOG function before each call to BGI routine, you can define your own coordinates-conversion function. Call `SET-COORDINATES!` with three functions as parameters, one extracting the device's X-coordinate from a point, one extracting the Y-coordinate, and the reverse procedure which returns the point corresponding to given device coordinates:

```
(define (my-x point)                      ; X-coord scaled from 0 to 1
  (round                                  ; Return a whole number
    (* (car point)                        ; Multiply the given X-coordinate...
       (car (get-max-xy)))))              ; ...by the maximum X-coordinate

(define (my-y point)                      ; Y-coord log: 1 (bottom) to 1000
  (round                                  ; Return a whole number
    (* (- 1 (log (cdr point) 1000))       ; Log of reversed y-coordinate...
       (cdr (get-max-xy)))))              ; ...times the maximum y-coordinate

(define (reverse-xy xy)                   ; Reverse procedure, used when BGI
  (let ((max-xy (get-max-xy)))            ; wants to return a coordinate
    (cons                                 ; Return a point (a pair)
      (/ (car xy) (car max-xy))
      (expt 1000 (- 1 (/ (cdr xy) (cdr max-xy)))))))

(set-coordinates! my-x my-y reverse-xy) ; Use the vertical-logarithmic system
```

Don't forget to call `SET-POINT?-!`, because `SET-COORDINATES!` won't call it for you (the example above assumes you've already typed in the `SET-POINT?-!` example).

Note that using this coordinate system, the same call to a function such as LINE-REL can have different results, depending on where the pen is:

```
(move-to '(0.5 . 1))
(line-rel '(0 . 10))    ; length = 1/3 of total height

(move-to '(0.5 . 100)) ; goes up into log scale...
(line-rel '(0 . 10))    ; length = 1/72 of total height
```

Of course, you don't need to assume that a point is a pair; why not a list of numbers, in a three- or four-dimensional space? Just write the functions that extract a projection out of a vector, and you're done!

Do you like Euclid? If you don't, you can also define how to calculate distances, almost the same way you defined the coordinates. `SET-DISTANCES!` accepts three parameters: the X distance extractor, the Y distance extractor and the unary distance extractor. These three functions receive the point from which the distance is to be calculated and the user-desired distance. X & Y distances are used for the following functions: `MOVE-REL`, `LINE-REL`, `ELLIPSE`, `FILL-ELLIPSE` and `SECTOR`, while the unary distance is used by `ARC`, `CIRCLE` and `PIE-SLICE` ("unary" means that the function receives distances as a number and returns a number; while X and Y distances receive a fictive point which is the displacement between two points).

All these coordinate-manipulation primitives return the previous version of all the procedures they change, so you can restore them later. In particular, SET-WORLD! returns all of seven procedures, in the order `POINT?, X, Y, REVERSE-XY, X-DIST, Y-DIST,` and `UNARY-DIST`; and all can be restored in one single call to `RESTORE-WORLD!`:

```
(let ((old-sys (set-world!  '(0 . 0) '(1 . 1)))) ; change and remember
  ...
  (restore-world! old-sys))
```

If you want to restore procedures by yourself, note the following example:

```
(let* ((old-sys (set-world!  '(0 . 0) '(1 . 1))) ; change and remember
 (old-point? (car old-sys))
 (old-coord (cdr old-sys))
```

```
(old-dist (cdddr old-coord)))
  ...
  (set-point?-!     old-point?)
  (set-coordinates! (car old-coord) (cadr old-coord) (caddr old-coord))
  (set-distances!   (car old-dist)  (cadr old-dist)  (caddr old-dist))
```

Always restore procedures in this order, since `SET-COORDINATES!` modifies the distance-functions to adapt them to the new given system.

5.5 The PS and SPY drivers

This part has been written by Larry Bartholdi.

One one the restrictions of BGI was that it could handle displays, but not printers. To overcome this limitation, some people developed "printer packages" for BGI. Although they derive from a real need, they are almost useless in that the force a complete re-write of the high-level code, linking with extra libraries, etc. By contrast, the solution I propose is a simple "plug-and-play": the driver should be recoginsed by any system using BGI that allows user-supplied drivers to be installed (see `INSTALL-USER-DRIVER`). The price to pay is, of course, that the driver produces only PostScript output. This postScript can then be send to a laser printer, inserted in a text document, or converted to another format or device. See for instance GNU's "ghostscript".

The drivers are written in C. This makes them very easy to change, while requiring very little extra space. An interface to the C code, written in assembly language is provided. Users are free to write their own BGI drivers using this system.

5.5.1 How to use SPY

SPY is really a very simple driver, that was written mostly to debug the C interfacing (whence the name). It has only one mode, 1 color (really two—black and white) and a fictive resolution of 100×100. All it does is write the graphics requests it receives to the standard output.

5.5.2 How to use PS

PS basically operates like SPY: it translates graphics requests to text, which it then writes to a file (or the standard output if none is specified). The environment variable "`PS$`" specifies on which file the output should go. Beware that the variable cannot be changed from within PcScheme. This is a strange behaviour I don't really care to deal with; but it is due to duplication of the environment.

PS accepts a wide variety of display modes, because the Postscript language is by nature pixel-independent while BGI is pixel-oriented; and the lines' width, points' size etc. are determined by the device's resolution. There are 6 normal modes, and 6 encapsulated modes. The former create output ready to send to a printer, while the latter are best suited to create images to be inserted in a document.

There is an eternal problem with color and display versus hardcopy. The colors are not changed, but the greytones are swapped. This is logical since a screen is black by default, while a blank sheet of paper is white. The driver accepts the same color requests as a Super-Vga card: 16 colors by default and a user-settable 256-entry palette. Here are the first preset 16 colors:

Index	**Color**	**Red**	**Green**	**Blue**
0	White	100%	100%	100%
1	Dark Blue	0%	0%	50%
2	Dark Green	0%	50%	0%
3	Dark Cyan	0%	50%	50%
4	Dark Red	50%	0%	0%
5	Dark Magenta	50%	0%	50%
6	Dark Brown	50%	50%	0%
7	Dark Grey	50%	50%	50%
8	Light Grey	75%	75%	75%
9	Blue	0%	0%	100%
10	Green	0%	100%	0%
11	Cyan	0%	100%	100%
12	Red	100%	0%	0%
13	Magenta	100%	0%	100%
14	Brown	100%	100%	0%
15	Black	0%	0%	0%

If your printer has no color capabilities, these values are converted to greytones by combining linearly the RGB percentages.

Mode	**Type**	**Resolution**	**Page Size [inches]**
0	Postscript	100×100	7×7
1	Postscript	320×200	11.2×7
2	Postscript	640×480	9.333×7
3	Postscript	1024×768	9.333×7
4	Postscript	1024×1024	7×7
5	Postscript	2048×2048	7×7
6	Encapsulated	100×100	7×7
7	Encapsulated	320×200	11.200×7
8	Encapsulated	640×480	9.333×7
9	Encapsulated	1024×768	9.333×7
10	Encapsulated	1024×1024	7×7
11	Encapsulated	2048×2048	7×7

An example of PS can be seen in the Hershey sample code. PS has been used there to insert sample displays in a TEX document.

There are no restrictions nor royalties associated to the use, distibution or modification of PS and SPY.

Chapter 6

PcScheme BGI primitives reference

This section is a list of recognized BGI primitives, by category. In each category, they are sorted by "logical" order, that is, the simplest is at the beginning and the most seldom used or complex at the end. Essential procedures are marked with a harpoon (↩) in the margin. Optional parameters are enclosed in square brackets ([ ]).

Color, video modes and some other types of parameters for which symbols are listed in this list can be entered either as symbol or as the corresponding number; numbers exist only for BGI compatibility, but you should only use symbols, as they are more "human". When a function such as `GET-TEXT-SETTINGS` is used, a symbol is returned. The only exceptions are `GET-COLOR` which returns a number to avoid confusion when re-mapping the palette, and `GET-GRAPH-MODE` because the corresponding symbol depends on the currently loaded driver and might be not available when using an user-installed driver. Equivalence tables can be found in the `BGI-ENVIRONMENT`.

6.1 The Graphics Control System

`(INIT-GRAPH` [*driver* [*graph-mode* [*path-for-BGI-files*]]]`)` ↩

Init-graph loads the BGI driver into memory, initialize all to the defaults values and turn the graphic device into graphic mode. *You cannot issue any graphic command prior to calling* `INIT-GRAPH`. In addition to these standard operations, PcScheme also reduces the *'CONSOLE* window to the bottom 4 lines of screen, i.e. calls `(SPLIT-SCREEN 4)`. Use `(FULL-SCREEN)` to use the whole text screen.

Driver can be any symbol returned by `INSTALL-USER-DRIVER`, or one of the following:

'DETECT	*'EGA*	*'IBM8514*	*'VGA*
'CGA	*'EGA64*	*'HERCMONO*	*'PC3270*
'MCGA	*'EGAMONO*	*'ATT400*	

'DETECT causes auto-detection of graphic hardware. It works only for all the original Borland drivers except Ibm-8514.

The *graph-mode* range depends of the BGI driver. It can be obtained using `the GET-MAX-MODE` primitive (see below). You can use the following symbols, or a valid number:

'CGA-C0	320 × 200, 4 colors	*'CGA-HI*	640 × 200, 2 colors
'CGA-C1	320 × 200, 4 colors	*'MCGA-C0*	320 × 200, 4 colors
'CGA-C2	320 × 200, 4 colors	*'MCGA-C1*	320 × 200, 4 colors
'CGA-C3	320 × 200, 4 colors	*'MCGA-C2*	320 × 200, 4 colors

[1] if 256K on board

'MCGA-C3	320×200, 4 colors	*'EGA-HI*	640×350, 16 colors, 2 pages
'MCGA-MED	640×200, 2 colors	*'EGA64-LO*	640×200, 16 colors
'MCGA-HI	640×480, 2 colors	*'EGA64-HI*	640×350, 16 colors
'ATT400-C0	320×200, 4 colors	*'EGAMONO-HI*	640×350, 2 colors, 2 pages[1]
'ATT400-C1	320×200, 4 colors	*'VGA-LO*	640×200, 16 colors, 2 pages
'ATT400-C2	320×200, 4 colors	*'VGA-MED*	640×350, 16 colors, 2 pages
'ATT400-C3	320×200, 4 colors	*'VGA-HI*	640×480, 16 colors
'ATT400-MED	640×200, 2 colors	*'HERCMONO-HI*	720×348, 2 colors, 2 pages
'ATT400-HI	640×400, 2 colors	*'PC3270-HI*	720×350, 2 colors
'EGA-LO	640×200, 16 colors, 4 pages	*'IBM8514-LO*	1024×768, 256 colors
		'IBM8514-HI	640×480, 256 colors

'CGA-C$\langle n \rangle$ are modes with same resolution but different palettes:

- Mode 0 allows *BACKGROUND, LIGHT-GREEN, LIGHT-RED* and *YELLOW*
- Mode 1 allows *BACKGROUND, LIGHT-CYAN, LIGHT-MAGENTA* and *WHITE*
- Mode 2 allows *BACKGROUND, GREEN, RED* and *BROWN*
- Mode 3 allows *BACKGROUND, CYAN, MAGENTA* and *LIGHT-GRAY*

This is true for *'MCGA-C*$\langle n \rangle$ and *'ATT400-C*$\langle n \rangle$ too.

The default values are:

driver	*'DETECT*
mode	Best colors, best resolution
path-for-BGI-files	`PCS-SYSDIR`, the directory containing `BOOTSTRP.APP`

(SET-WRITE-MODE *wmode*)

This global setting allows you to choose what to do when drawing overwrite a previously plotted object.

Wmode can be one of the following: ↩

'COPY (this is the default)
'XOR

This function currently works only for lines, rectangles, polygons and text. `PUT-IMAGE` has his own setting, with other possible modes.

(RESTORE-CRT-MODE) ↩

This brings the display back to the text mode when `INIT-GRAPH` was used, but doesn't unload BGI drivers from memory. It is useful with `SET-GRAPH-MODE` to toggle between graphic and text mode.

(SET-GRAPH-MODE [*mode*]) ↩

This changes the current video mode to the one given as parameter. Correct values are listed under `INIT-GRAPH`, or can be obtained by a call to `GET-MAX-MODE`. `SET-GRAPH-MODE` can be used with `RESTORE-CRT-MODE` to toggle between graphics and text.

The default value for *mode* is the previously used graph mode.

(CLOSE-GRAPH) ↩

This turns the video adapter back to text mode, and unloads the driver from memory. *You should always use CLOSE-GRAPH at the end of your graph programs,* or you have risks to load another copy of BGI driver into memory each time you start it, which would cause sooner or later a fatal out-of-memory error.

`(GRAPH-DEFAULTS)`

This resets all graphics settings to their default values:

- Sets the viewport to entire screen
- Moves the pen to upper left corner
- Sets the default colors and palette
- Sets all default style, patterns, text fonts and justification

`(DETECT-GRAPH)`

Tries to recognize the video hardware present in the computer, and returns the detected adapter with the best mode available. See the list of adapters and modes under `INIT-GRAPH`. The returned driver and mode (in a pair) can be used to find out what would be the result of a call to `(INIT-GRAPH 'DETECT)`. User drivers and *'IBM8514* cannot be auto-detected.

`(GET-MODE-RANGE` [*driver*]`)`

Returns the range of values acceptable as graphics mode for a given video hardware. If *driver* is not valid, returns *'(-1 . -1)*; otherwise returns a pair made of the smallest and the largest mode.

This works with Borland "factory" drivers only. Preferably use `GET-MAX-MODE`.

The default value for *driver* is the current driver.

`(GET-GRAPH-MODE)`

Returns the current graphic mode number. Can be used with `SET-GRAPH-MODE` to remember and restore the video mode from a session to the other, but should be used carefully in order to maintain device-independence of programs.

`(INSTALL-USER-DRIVER` *name*`)`

Installs the driver whose name is specified as user BGI driver. The HP95LX driver should be used this way, since it is not part of the standard BGI drivers. `INSTALL-USER-DRIVER` establishes a link between the driver and the list of known drivers. It returns a symbol, made of name, which can then be used when calling `INIT-GRAPH`. The argument *name* is a string.

`(INSTALL-USER-FONT` *name*`)`

Installs the font whose name is specified as user font, and returns its symbolic equivalent in same manner as `INSTALL-USER-DRIVER`.

6.2 Drawing

`(LINE` *start-point end-point*`)` ↩

Draws a line using current color, line-style and write-mode. *Start-point* and *end-point* are points for the current coordinate system, i.e. pairs by default.

`(RECTANGLE` *upper-left-point lower-right-point*`)` ↩

Draws an empty rectangle using current color, line-style, write-mode.

`(DRAW-POLY` *list-of-points*`)` ↩

Draws a polygon, i.e. lines from points to points. If you want the polygon to be closed, you need to put the first point again at the end of the list.

`(CIRCLE` *center-point radius*`)` ↩

Draws a circle using current color and thickness (part of line-style). If your circles look oval, use `SET-ASPECT-RATIO` to correct them.

`(ARC` *center-point start-angle end-angle radius*`)`

Draws an arc of circle using color and thickness (see `CIRCLE`). Angles are given in degrees, using whole numbers. `ARC` returns the coordinates of the arc as `GET-ARC-COORDS` would have do it.

`(ELLIPSE` *center-point start-angle end-angle distances*`)`

Exactly the same as `ARC`, but distances is a pair of X and Y radius. To draw a whole ellipse, let *start-angle* $= 0$ and *end-angle* $= 360$.

`(GET-ARC-COORDS)`

Returns a list of three points: the center-point of the last arc drawn, the starting-point and the ending-point (i.e. 3 pairs). This command is useful if you want to join lines and arc curves.

`(SET-ASPECT-RATIO` *factor*`)`

Factor is a pair of integers p and q representing a rational number, p/q, a coefficient by which the radius is multiplied to obtain the y-radius when drawing a circle or an arc. Use this function either to adjust circles when your video adapter makes circles oval, or to draw ellipses of given proportion but of different sizes.

`(GET-ASPECT-RATIO)`

Returns the factor as described in `SET-ASPECT-RATIO`.

`(SET-LINE-STYLE` *line-style user-pattern thickness*`)` ↩

Sets the line drawing parameters: style and width. If *line-style* is *'USER-BIT*, *user-pattern* is a 16-bit integer specifying the pattern used to draw straight lines. For instance, a user-defined pattern of `#b1110001110001010` would draw “”.

Line-style is one of the following:

'SOLID	*'CENTER*	*'USER-BIT*
'DOTTED	*'DASHED*	

Thickness can be any positive integer, and in particular:

'NORMAL which is 1
'THICK which is 3

Other values are not illegal, but do not necessarily produce different results.

`(GET-LINE-SETTINGS)`

Returns the list made of the three settings described in `SET-LINE-STYLE`.

`(MOVE-TO` *point*`)`

Moves the "pen" to the given position, without changing anything on the screen.

`(LINE-TO` *point*`)`

Draws a line from the current "pen" position to *point*, using the current line style, color and thickness

`(MOVE-REL` *distances*`)`

Same as `MOVE-TO`, except the move is done relatively to the current pen position.

`(LINE-REL` *distances*`)`

You make a soup with `LINE-TO` and `MOVE-REL` and you'll get it right.

6.3 Filling

`(FLOOD-FILL` *start-point stop-color*`)` ↩

Fills a region of the screen starting from start-point until *stop-color* is encountered. When used with open shapes, the whole screen might get filled.

`FLOOD-FILL` is not yet supported for IBM-8514 and HP95LX drivers. Where possible, avoid using `FLOOD-FILL`.

`(BAR` *upper-left-point lower-right-point*`)` ↩

Draws a solid 2-dimensional bar without outlet, using the current fill pattern and fill color.

`(BAR-3D` *upper-left-point lower-right-point depth top?*`)`

Draws a solid 3-dimensional bar with outlets, using the current fill pattern and fill color for the front face, and current line width and color for outlet. *Depth* is the 3rd dimension, given in the X-dimension coordinate system. *Top?* is `#T` if a top should be drawn, and `#F` if not. `#T` and `#F` can be replaced by 1 and 0). This function is very useful for drawing bar charts.

`(FILL-POLY` *list-of-points*`)`

Draws a filled polygon using the current fill pattern and color. See `DRAW-POLY`.

`(FILL-ELLIPSE` *center-point distances*`)` ↩

Draws a filled ellipse using the current fill pattern and color.

`(PIE-SLICE` *center-point start-angle end-angle radius*`)`

Fills a slice of pie. Angles are given in degrees. Use `SET-ASPECT-RATIO` if your pie looks sad.

(SECTOR *center-point start-angle end-angle distances*)

Draws a slice of a filled ellipse using current fill pattern and fill color (most general function).

(SET-FILL-STYLE *fill-style color*) ↩

Sets the current fill pattern to one of the defined pattern and selects the fill color. *Fill-style* can be one of the following:

'EMPTY	all background color
'SOLID	all fill color
'LINE	continuous
'LTSLASH	light
'SLASH	thick
'LTBKSLASH	light
'BKSLASH	thick
'HATCH	hatch
'XHATCH	X-hatch
'INTERLEAVE	50% grey
'CLOSE-DOT	20% grey
'WIDE-DOT	10% grey
'USER-FILL	

Don't use (SET-FILL-STYLE *'USER-FILL* ...), but use SET-FILL-PATTERN instead, or you might get unpredictable results.

For a table of color constants, see SET-COLOR below.

(SET-FILL-PATTERN *fill-pattern color*) ↩

This call is similar to SET-FILL-STYLE, except that it sets a user-defined pattern instead of a predefined one. A fill-pattern is an 8x8 matrix used to fill surfaces; at the PcSchemelevel, it is a list of 8-bit integers, each coding one row. Normally there will be 8 elements in the list but if you put less, PCS will assume you want to repeat periodically the given ones. For example, the chain pattern

```
(SET-FILL-PATTERN '(#b01000100
                    #b00111000
                    #b01000100
                    #b01000100
                    #b01000100
                    #b00111000
                    #b01000100
                    #b01000100) 'BLUE)
```

will produce the same result as

```
(SET-FILL-PATTERN '(#b00111000
                    #b01000100
                    #b01000100
                    #b01000100) 'BLUE)
```

(GET-FILL-SETTINGS)

This returns a pair containing the standard fill style number and the fill color. If the fill style number is 12 (*'USER-FILL*), you can issue a call to GET-FILL-PATTERN to get the exact pattern.

(GET-FILL-PATTERN)

This returns a string containing the fill pattern.

6.4 Bitmapping

(CLEAR-DEVICE) ↩

Fills the whole screen with the background color and moves the pen to the upper-left corner.

(PUT-PIXEL *point color*) ↩

Plots a single pixel at coordinates specified by point, using the desired color (see **SET-COLOR** below for color names).

(GET-PIXEL *point*) ↩

Returns the color number of the selected pixel.

(GET-IMAGE *upper-left-point lower-right-point*) ↩

Returns an image-string containing all bitmap data for the rectangle enclosed between the two given points. Don't try to look at too big strings of this type, or you might have to wait for a long time until you reach the end, especially if there is a bell every two chars...

You should use **IMAGE-SIZE** to figure out how much memory a bitmap needs, and never allocate an image of 32Kb or more.

(PUT-IMAGE *new-upper-left-point image-string put-mode*) ↩

Rewrites (very quickly!) the whole image saved in image-string, with the upper left corner at *new-upper-left-point*. You can use various displaying methods:

'COPY	*'XOR*	*'OR*	*'AND*	*'NOT*

This function can be used to animate a small shape on the screen.

(IMAGE-SIZE *upper-left-point lower-right-point*)

Returns the size in bytes required to store the bitmap enclosed between the two given points.

(SET-VIEWPORT *upper-left-point lower-right-point clip?*)

Translates the coordinate system so that the coordinates of screen upper-left corner before a call are the same as the coordinates of upper-left-point after the call; moves the pen to this point; if *clip?* is **#T** or 1, clip all incoming function calls to the rectangle between the two points (i.e. no drawing will occur outside of this zone).

The redefinition of viewport is always relative to the whole screen, using the global (user- or not) coordinates system.

(CLEAR-VIEWPORT)

Fills the active viewport with the background color and moves the pen to the upper left corner of the viewport.

(GET-VIEW-SETTINGS)

Returns the list of **SET-VIEWPORT**'s three parameters (where *clip?* is either 1 or 0).

(SET-ACTIVE-PAGE *page*)

For graphic adapters which support more than one page (see the table at **INIT-GRAPH**), this functions allows one to choose on which page the next drawings should have effect.

It doesn't have to be the visual page, so one can draw off-screen and then bring the prepared page using just one command (see **SET-VISUAL-PAGE**).

(SET-VISUAL-PAGE *page*)

For graphic adapters which support more than one page this functions allows one to choose which page is to be displayed on the screen.

It doesn't have to be the active page, so one can draw off-screen and then bring the prepared page using just one command (see **SET-ACTIVE-PAGE**).

6.5 Writing text

(OUT-TEXT-XY *start-point text-string*) ↩

Writes *text-string* starting from *start-point* according to current justification (see **SET-TEXT-JUSTIFY**) and text style (see **SET-TEXT-STYLE**). The current color and write mode are used.

(OUT-TEXT *text-string*)

Writes *text-string* starting at the current pen position, according to current justification, style, color and write mode. If the current direction is *'HORIZ* and the text justification is *'LEFT*, the pen is moved to the right of the text; otherwise, it is left unchanged.

(SET-TEXT-STYLE *font direction size*) ↩

Sets text appearance properties. *Font* can be either a number returned by **INSTALL-USER-FONT** or one of the following, where *'DEFAULT* is the only bit-mapped font:

'DEFAULT	*'SANS-SERIF*	*'SIMPLEX*	*'EUROPEAN*
'TRIPLEX	*'GOTHIC*	*'TRIPLEX-SCR*	*'BOLD*
'SMALL	*'SCRIPT*	*'COMPLEX*	

Direction is *'HORIZ* or *'VERT*.

Size is either a number from 1 to 10, or 0 for the default size. The default size will produce a size of 4, but will also enable user character sizes (see **SET-USER-CHAR-SIZE** below).

(SET-TEXT-JUSTIFY *horiz-just vert-just*) ↩

Choose where to align the text. *Horiz-just* is one of the following:

'LEFT *'CENTER* *'RIGHT*

Vert-just is one of the following:

'BOTTOM *'CENTER* *'TOP*

For example, *'LEFT* means that the starting-point is on the top left edge of the text.

(GET-TEXT-SETTINGS)

Returns a list of the five text settings: font, direction, size, horizontal and vertical justification.

(SET-USER-CHAR-SIZE *x-ratio y-ratio*)

When *font-size* is set to 0, this allows you to give the font any width and height. The font's X and Y sizes are multiplied by the x- and y-ratios. Ratios are rational numbers, i.e. pairs of integers.

(TEXT-SIZE *text-string*) ↩

Returns a pair containing the width and height of *text-string* using the current font. It doesn't care about text direction. The height is independent of the character written, i.e. an "I" has the same height as an "x".

6.6 Using Color

(SET-COLOR *color*) ↩

Sets the current color for all drawing functions. If you are have 16 or more colors, *color* can be one of the following:

Symbol	**Equivalent number**
'BLACK	0
'BLUE	1
'GREEN	2
'CYAN	3
'RED	4
'MAGENTA	5
'BROWN	6
'LIGHT-GRAY	7
'DARK-GRAY	8
'LIGHT-BLUE	9
'LIGHT-GREEN	10
'LIGHT-CYAN	11
'LIGHT-RED	12
'LIGHT-MAGENTA	13
'YELLOW	14
'WHITE	15

If you are using CGA, MCGA or ATT-400 four-color modes, you should use instead the following constants, depending on your palette:

0	*'BACKGROUND*	*'CGA-LIGHT-GREEN*	*'CGA-LIGHT-RED*	*'CGA-YELLOW*
1	*'BACKGROUND*	*'CGA-LIGHT-CYAN*	*'CGA-LIGHT-MAGENTA*	*'CGA-WHITE*
2	*'BACKGROUND*	*'CGA-GREEN*	*'CGA-RED*	*'CGA-BROWN*
3	*'BACKGROUND*	*'CGA-CYAN*	*'CGA-MAGENTA*	*'CGA-LIGHT-GRAY*

You can change the effect of SET-COLOR using SET-PALETTE. With CGA-style adapters, only the first color (i.e. black, the background color) can be changed (it is best to use SET-BK-COLOR).

If you totally re-map the palette with a EGA or VGA adapter, use SET-COLOR with numbers rather than with symbols, since it doesn't make programs very clear when (SET-COLOR *'BLUE*) turns color to green for example...

(SET-BK-COLOR *color*) ↩

Sets the background color to the specified color. *Color* can be any of the following (for CGA-likes too):

Symbol	Equivalent number
'BLACK	0
'BLUE	1
'GREEN	2
'CYAN	3
'RED	4
'MAGENTA	5
'BROWN	6
'LIGHT-GRAY	7
'DARK-GRAY	8
'LIGHT-BLUE	9
'LIGHT-GREEN	10
'LIGHT-CYAN	11
'LIGHT-RED	12
'LIGHT-MAGENTA	13
'YELLOW	14
'WHITE	15

On CGA-ish and EGA systems, `SET-BK-COLOR` only changes the entry 0 of the palette.

If you totally re-map the palette with a EGA or VGA adapter, use `SET-BK-COLOR` with numbers rather than symbols.

(GET-COLOR)

Returns current color number (not symbol, since a palette change would mix-up everything).

(GET-BK-COLOR)

Returns current background color number.

(GET-MAX-COLOR) ↩

Returns the number of the greatest acceptable color for current video mode (i.e. #-of-colors minus 1).

(SET-PALETTE *entry color*)

Re-map the given entry of the palette to the specified color. It is useful for EGA and VGA (for CGA, it works only with entry 0, i.e. background color). *Entry* is an integer between 0 and `(GET-PALETTE-SIZE)`, and *color* is an integer between 0 and a constant depending on the hardware device. The following constants are defined:

Symbol	Equivalent number	
'EGA-BLACK	0	(#b000000)
'EGA-BLUE	1	(#b000001)
'EGA-GREEN	2	(#b000010)
'EGA-CYAN	3	
'EGA-RED	4	(#b000100)
'EGA-MAGENTA	5	
'EGA-LIGHT-GRAY	7	
'EGA-BROWN	20	
'EGA-DARK-GRAY	56	
'EGA-LIGHT-BLUE	57	(#b001001)
'EGA-LIGHT-GREEN	58	(#b010010)
'EGA-LIGHT-CYAN	59	
'EGA-LIGHT-RED	60	(#b100100)
'EGA-LIGHT-MAGENTA	61	
'EGA-YELLOW	62	
'EGA-WHITE	63	(#b111111)

These are the values of the standard EGA and VGA palette. With these adapters, the following code would be equivalent to `(SET-ALL-PALETTE (GET-DEFAULT-PALETTE))`:

```
(SET-PALETTE 'BLACK 'EGA-BLACK)
(SET-PALETTE 'BLUE 'EGA-BLUE)
(SET-PALETTE 'RED 'EGA-RED)
(SET-PALETTE 'GREEN 'EGA-GREEN)
...
(SET-PALETTE 'WHITE 'EGA-WHITE)
```

`SET-PALETTE` doesn't works with the IBM-8514. Use `SET-RGB-PALETTE` instead.

(SET-RGB-PALETTE *entry red green blue*)

This function has the same effect as `SET-PALETTE`, except that it works only with the VGA 256K, the IBM-8514, and the SUPER-VGA adapters. *Red*, *green* and *blue* are the strength of each color component. The 6 most significants bits of the least significant byte are used.

(SET-ALL-PALETTE *color-list*)

Color-list is a list of length `(GET-PALETTE-SIZE)`. This remaps all colors at the same time. There is no "entry" parameter as there was with `SET-PALETTE`, since all colors are re-mapped in the normal order.

(GET-PALETTE)

Returns a color-list corresponding to current palette.

(GET-DEFAULT-PALETTE)

Returns a list with the default palette values.

(GET-PALETTE-SIZE)

Returns the number of entries in the palette.

6.7 Miscellaneous queries

(GET-MAX-XY) ↩

Returns the maximum screen coordinates *using pixel dimensions.* (1 unit = 1 pixel). It doesn't depend of **SET-WORLD!** settings, so it can be used to determine the actual resolution of the current video mode, in order to write your own coordinate conversion routine or to scan the screen pixel per pixel.

(GET-XY)

Returns the current pen location.

(GET-DRIVER-NAME) ↩

Returns the name of the currently loaded driver, without path and without its ".BGI" extension. When no driver is loaded, returns "". Can be used to know if a call to **INIT-GRAPH** is necessary or not. A good startup code would be:

```
(if (= (GET-DRIVER-NAME) "")
    (INIT-GRAPH)
    (SET-GRAPH-MODE))
```

Thus you can abort your program as often as you want, and restart it without filling the memory with copies of the graphics driver.

(GET-MODE-NAME *mode*)

Returns a string corresponding to the real name of the specified video mode.

(GET-MAX-MODE)

Returns the maximum possible value for the video mode for the BGI driver currently loaded in memory. The minimum value is 0.

(GRAPH-ERROR-MSG *error-id*)

Returns the complete text message corresponding to a detected BGI error (actually used in the debugger; you shouldn't need it, except if you want to handle yourself BGI errors).

(GRAPH-RESULT)

Returns the error-id corresponding to last BGI call. Use it with **GRAPH-ERROR-MSG** to signal an error (already done for you).