

De Re BASIC!

Version 1.91

March 14, 2017



Table of Contents

Changes in this Version	24
About the Title, De Re BASIC!	25
About the Cover Art	25
Credits	25
Technical Editor	25
Getting BASIC!	25
BASIC! Forum	25
BASIC! Tutorial	25
BASIC! Operation	26
Permissions	26
Editor	26
Editing the Program	26
Multiple Commands on a Line	27
Line Continuation	27
# - Format Line	27
Menus	28
Run	34
Menu	35
Crashes	35
A BASIC! Program	35
Command Description Syntax	36
Upper and Lower Case	36
<nexp>, <sexp> and <lexp>	36
<nvar>, <svar> and <lvar>	36
Array[] and Array\$[]	36
Array[{<start>,<length>}] and Array\$[{<start>,<length>}]	36
{something}	36
{ A B C }	36
X,	37
{n}	37

<statement>.....	37
Optional Parameters.....	37
Numbers.....	37
Strings	38
Variables	38
Variable Names.....	38
Variable Types.....	39
Scalar and Array Variables	39
Scalars	39
Arrays	39
Array Segments.....	40
Array Commands.....	40
Data Structures and Pointers in BASIC!	43
What is a Pointer.....	43
Lists	45
List Commands.....	45
Bundles.....	47
Bundle Auto-Create	48
Bundle Commands.....	48
Stacks	50
Stack Commands.....	51
Queues.....	51
Comments.....	52
! - Single Line Comment.....	52
Rem - Single Line Comment (legacy).....	52
!! - Block Comment	52
% - Middle of Line Comment	52
Expressions.....	52
Numeric Expression <nexp>.....	52
Numeric Operators <noperator>	53
Numeric Expression Examples	53
Pre- and Post-Increment Operators.....	53

String Expression <sexp>	53
Logical Expression <lexp>	54
Logical Operators	54
Examples of Logical Expressions	54
Parentheses	54
Assignment Operations.....	55
Let	55
OpEqual Assignment Operations	55
Math Functions	56
BOR(<nexp1>, <nexp2>)	56
BAND(<nexp1>, <nexp2>)	56
BXOR(<nexp1>, <nexp2>)	56
BNOT(<nexp>).....	56
ABS(<nexp>).....	56
SGN(<nexp>)	56
RANDOMIZE({<nexp>})	57
RND().....	57
MAX(<nexp>, <nexp>).....	57
MIN(<nexp>, <nexp>)	57
CEIL(<nexp>)	57
FLOOR(<nexp>)	57
INT(<nexp>).....	57
FRAC(<nexp>).....	57
MOD(<nexp1>, <nexp2>).....	58
ROUND(<value_nexp>{, <count_nexp>{, <mode_sexp>}})	58
SQR(<nexp>)	59
CBRT(<nexp>).....	59
LOG(<nexp>)	59
LOG10(<nexp>)	59
EXP(<nexp>)	59
POW(<nexp1>, <nexp2>)	59
HYPOT(<nexp_x>, <nexp_y>).....	59

PI()	59
SIN(<nexp>)	59
COS(<nexp>)	59
TAN(<nexp>)	59
SINH(<nexp>)	59
COSH(<nexp>)	60
ASIN(<nexp>)	60
ACOS(<nexp>)	60
ATAN(<nexp>)	60
ATAN2(<nexp_y>, <nexp_x>)	60
TODEGREES(<nexp>)	60
TORADIANS(<nexp>)	60
VAL(<sexp>)	60
IS_NUMBER(<sexp>)	60
LEN(<sexp>)	61
HEX(<sexp>)	61
OCT(<sexp>)	61
BIN(<sexp>)	61
SHIFT(<value_nexp>, <bits_nexp>)	61
ASCII(<sexp>{, <index_nexp>})	61
UCODE(<sexp>{, <index_nexp>})	61
IS_IN(<sub_sexp>, <base_sexp>{, <start_nexp>})	61
STARTS_WITH(<sub_sexp>, <base_sexp>{, <start_nexp>})	62
ENDS_WITH(<sub_sexp>, <base_sexp>)	62
GR_COLLISION(<object_1_nvar>, <object_2_nvar>)	62
BACKGROUND()	62
Time Functions	63
CLOCK()	63
TIME()	63
TIME(<year_exp>, <month_exp>, <day_exp>, <hour_exp>, <minute_exp>, <second_exp>)	63
String Functions	63
GETERROR\$()	63

CHR\$(<nexp>, ...)	64
LEFT\$(<sexp>, <count_nexp>)	64
MID\$(<sexp>, <start_nexp>{, <count_nexp>})	64
RIGHT\$(<sexp>, <count_nexp>)	65
REPLACE\$(<sexp>, <argument_sexp>, <replace_sexp>)	65
TRIM\$(<sexp>{, <test_sexp>})	65
LTRIM\$(<sexp>{, <test_sexp>})	65
RTRIM\$(<sexp>{, <test_sexp>})	65
WORD\$(<source_sexp>, <n_nexp> {, <test_sexp>})	66
ENCODE\$(<type_sexp>, {<qualifier_sexp>, <source_sexp>})	66
DECODE\$(<type_sexp>, {<qualifier_sexp>, <source_sexp>})	67
ENCODE\$(<charset_sexp>, <source_sexp>)	68
DECODE\$(<charset_sexp>, <buffer_sexp>)	68
STR\$(<nexp>)	69
LOWER\$(<sexp>)	69
UPPER\$(<sexp>)	69
VERSION\$()	69
INT\$(<nexp>)	69
HEX\$(<nexp>)	69
OCT\$(<nexp>)	69
BIN\$(<nexp>)	69
USING\$({<locale_sexp>, <format_sexp> {, <exp>}...)	70
Locale expression	70
Format expression	70
Integer values	74
FORMAT_USING\$(<locale_sexp>, <format_sexp> {, <exp>}...)	74
FORMAT\$(<pattern_sexp>, <nexp>)	74
Notes	75
Examples	75
User-Defined Functions	75
Variable Scope	76
Data Structures in User-Defined Functions	76

Commands	76
Fn.def name name\$({nvar} {svar} Array[] Array\$[], ... {nvar} {svar} Array[] Array\$[])	76
Fn.rtn <sexp> <nexp>	77
Fn.end	78
Call <user_defined_function>	78
Program Control Commands	78
If / Then / Else / Elseif / Endif	78
If / Then / Else	79
For - To - Step / Next	79
F_N.continue	80
F_N.break	80
While <lexp> / Repeat	80
W_R.continue	81
W_R.break	81
Do / Until <lexp>	81
D_U.continue	81
D_U.break	82
Labels, GOTO, GOSUB, and RETURN: Traditional BASIC	82
Label	82
GoTo <label>	83
GoTo <index_nexp>, <label>	83
GoSub <label> / Return	83
GoSub <index_nexp>, <label>... / Return	83
Using Source Code from Multiple Files	84
Include FilePath	84
Run <filename_sexp>{, <data_sexp>}	85
Program.info <nexp> <nvar>	85
Switch Commands	86
Nesting Switch Operations	87
Sw.begin <exp>	87
Sw.case <exp>,	88
Sw.case <op><exp>	88

Sw.break.....	88
Sw.default	88
Sw.end.....	88
Interrupt Labels (Event Handlers).....	88
All Interrupt Labels.....	89
OnError:	89
OnConsoleTouch:.....	90
ConsoleTouch.resume	90
OnBackKey:	90
Back.resume.....	90
OnMenuKey:	90
MenuKey.resume	91
OnKeyPress:	91
Key.resume	91
OnLowMemory:	91
LowMemory.resume.....	91
End{ <msg_sexp>}.....	91
Exit.....	91
READ – DATA – RESTORE Commands	91
Read.data <number> <string>{,<number> <string>...,<number> <string>}	91
Read.next <var>,	92
Read.from <nexp>.....	92
Debug Commands.....	92
Debug.on.....	92
Debug.off	92
Debug.echo.on	92
Echo.on	93
Debug.echo.off.....	93
Echo.off	93
Debug.print	93
Debug.dump.scalars.....	93
Debug.dump.array Array[]	93

Debug.dump.bundle <bundlePtr_nexp>	93
Debug.dump.list <listPtr_nexp>	93
Debug.dump.stack <stackPtr_nexp>	93
Debug.show.scalars	93
Debug.show.array Array[]	94
Debug.show.bundle <bundlePtr_nexp>	94
Debug.show.list <listPtr_nexp>	94
Debug.show.stack <stackPtr_nexp>	94
Debug.watch var,	94
Debug.show.watch	94
Debug.show.program	94
Debug.show	95
Fonts.....	95
Font.load <font_ptr_nvar>, <filename_sexp>	95
Font.delete {<font_ptr_nexp>}	95
Font.clear	95
Console I/O	96
Output Console	96
Print {<exp> {, ;}}	96
? {<exp> {, ;}}	96
Print with User-Defined Functions.....	97
Cls.....	97
Console.front	97
Console.line.count <count_nvar>	97
Console.line.text <line_nexp>, <text_svar>	97
Console.line.touched <line_nvar> {, <press_lvar>}	98
Console.save <filename_sexp>	98
Console.title { <title_sexp>}	98
User Input and Interaction.....	98
Dialog.message {<title_sexp>}, {<message_sexp>}, <sel_nvar> {, <button1_sexp>{, <button2_sexp>{, <button3_sexp>}}}	98
Dialog.select <sel_nvar>, <Array\$[]> <list_nexp> {,<title_sexp>}	99

Input {<prompt_sexp>}, <result_var>{, {<default_exp>}{, <canceled_nvar>}}.....	100
Inkey\$ <svar>	100
Popup <message_sexp> {{, <x_nexp>}{, <y_nexp>}{, <duration_lexp>}}	101
Select <sel_nvar>, <Array\$[]> <list_nexp> {,<title_sexp> {, <message_sexp> } } {,<press_lvar> } ..	101
Text.input <svar>{, { <text_sexp> } , <title_sexp> }	101
TGet <result_svar>, <prompt_sexp> {, <title_sexp> }	102
Kb.hide	102
Kb.show	102
Kb.toggle	103
Kb.showing <lvar>	103
OnKbChange:	103
Kb.resume	103
The Soft Keyboard and the BACK Key	103
Working with Files.....	104
Paths Explained – a beginner’s guide	104
Paths in BASIC!	104
Paths Outside of BASIC!	105
Paths and Case-sensitivity.....	105
Mark and Mark Limit.....	106
File Commands.....	106
File.delete <lvar>, <path_sexp>	106
File.dir <path_sexp>, Array\$[] {,<dirmark_sexp>}	107
File.exists <lvar>, <path_sexp>	107
File.mkdir <path_sexp>.....	107
File.rename <old_path_sexp>, <new_path_sexp>	107
File.root <svar>	108
File.size <size_nvar>, <path_sexp>	108
File.type <type_svar>, <path_sexp>	108
Text File I/O.....	108
Text.open {r w a}, <file_table_nvar>, <path_sexp>	108
Text.close <file_table_nexp>	109
Text.readln <file_table_nexp> {,<svar>}... ..	109

Text.writeln <file_table_nexp>, <parms same as Print>	110
Text.eof <file_table_nexp>, <lvar>	110
Text.position.get <file_table_nexp>, <position_nvar>	110
Text.position.set <file_table_nexp>, <position_nexp>.....	110
Text.position.mark {{<file_table_nexp>},{, <marklimit_nexp>}}	111
GrabURL <result_svar>, <url_sexp>{, <timeout_nexp>}.....	111
GrabFile <result_svar>, <path_sexp>{, <unicode_flag_lexp>}.....	111
Byte File I/O.....	112
Byte.open {r w a}, <file_table_nvar>, <path_sexp>.....	112
Byte.close <file_table_nexp>	112
Byte.read.byte <file_table_nexp> {,<nvar>}...	112
Byte.write.byte <file_table_nexp> {{,<nexp>}...{,<sexp>}}	113
Byte.read.number <file_table_nexp> {,<nvar>}...	113
Byte.write.number <file_table_nexp> {,<nexp>}.....	113
Byte.read.buffer <file_table_nexp>, <count_nexp>, <buffer_svar>	114
Byte.write.buffer <file_table_nexp>, <buffer_sexp>	114
Byte.eof <file_table_nexp>, <lvar>	114
Byte.position.get <file_table_nexp>, <position_nvar>.....	114
Byte.position.set <file_table_nexp>, <position_nexp>	114
Byte.position.mark {{<file_table_nexp>},{, <marklimit_nexp>}}	115
Byte.truncate <file_table_nexp>,<length_nexp>	115
Byte.copy <file_table_nexp>,<output_file_sexp>	115
ZIP File I/O	115
Zip.count <path_sexp>, <nvar>.....	116
Zip.dir <path_sexp>, Array\$[] {,<dirmark_sexp>}.....	116
Zip.open {r w a}, <file_table_nvar>, <path_sexp>	116
Zip.close <file_table_nexp>	116
Zip.read <file_table_nexp> ,<buffer_svar>, <file_name_sexp>	116
Zip.write <file_table_nexp> ,<buffer_sexp>, <file_name_sexp>	117
HTML.....	117
Introduction	117
HTML Commands.....	117

Html.open {<ShowStatusBar_lexp> {, <Orientation_nexp>}}	117
Html.orientation <nexp>	118
Html.load.url <file_sexp>	118
Html.load.string <html_sexp>	118
Html.post <url_sexp>, <list_nexp>	119
Html.get.datalink <data_svar>	119
Html.go.back	120
Html.go.forward	120
Html.close	120
Html.clear.cache	120
Html.clear.history	120
Related Commands	120
Browse <url_sexp>	120
Http.post <url_sexp>, <list_nexp>, <result_svar>	120
TCP/IP Sockets	121
TCP/IP Client Socket Commands	121
Socket.client.connect <server_sexp>, <port_nexp> {, <wait_lexp> }	121
Socket.client.status <status_nvar>	122
Socket.client.server.ip <svar>	122
Socket.client.read.line <line_svar>	122
Socket.client.read.ready <nvar>	122
Socket.client.read.file <file_nexp>	122
Socket.client.write.line <line_sexp>	122
Socket.client.write.bytes <sexp>	123
Socket.client.write.file <file_nexp>	123
Socket.client.close	123
TCP/IP Server Socket Commands	123
Socket.myIP <svar>	123
Socket.myIP <array\$[]>{, <nvar> }	123
Socket.server.create <port_nexp>	123
Socket.server.connect {<wait_lexp>}	123
Socket.server.status <status_nvar>	124

Socket.server.read.line <svar>	124
Socket.server.read.ready <nvar>	124
Socket.server.write.line <line_sexp>	124
Socket.server.write.bytes <sexp>	124
Socket.server.write.file <file_nexp>	125
Socket.server.read.file <file_nexp>	125
Socket.server.disconnect	125
Socket.server.close	125
Socket.server.client.ip <nvar>	125
FTP Client	125
Ftp.open <url_sexp>, <port_nexp>, <user_sexp>, <pw_sexp>	125
Ftp.close	125
Ftp.put <source_sexp>, <destination_sexp>	125
Ftp.get <source_sexp>, <destination_sexp>	126
Ftp.dir <list_nvar> {,<dirmark_sexp>}.	126
Ftp.cd <new_directory_sexp>	126
Ftp.rename <old_filename_sexp>, <new_filename_sexp>	126
Ftp.delete <filename_sexp>	126
Ftp.rmdir <directory_sexp>	126
Ftp.mkdir <directory_sexp>	127
Bluetooth	127
Bt.open {0 1}	127
Bt.close	128
Bt.connect {0 1}	128
Bt.disconnect	128
Bt.reconnect	128
Bt.status {{<connect_var>},{, <name_svar>},{, <address_svar>}}	128
Bt.write {<exp> {, ;}}	129
Bt.read.ready <nvar>	129
OnBtReadReady:	129
Bt.onReadReady.resume	129
Bt.read.bytes <svar>	130

Bt.device.name <svar>	130
Bt.set.UUID <sexp>	130
Communication: Phone and Text	130
Email.send <recipient_sexp>, <subject_sexp>, <body_sexp>	130
MyPhoneNumber <svar>	130
Phone.call <sexp>	130
Phone.dial <sexp>	131
Phone.rcv.init	131
Phone.rcv.next <state_nvar>, <number_svar>	131
Sms.send <number_sexp>, <message_sexp>	131
Sms.rcv.init	131
Sms.rcv.next <svar>	131
Time and Timers	132
Time and TimeZone Commands	132
Time {<time_nexp>}, Year\$, Month\$, Day\$, Hour\$, Minute\$, Second\$, WeekDay, isDST	132
TimeZone.set { <tz_sexp> }	133
TimeZone.get <tz_svar>	133
TimeZone.list <tz_list_pointer_nexp>	133
Timer Interrupt and Commands	133
Timer.set <interval_nexp>	133
OnTimer:	133
Timer.resume	133
Timer.clear	134
Sample Code	134
Clipboard	134
Clipboard.get <svar>	134
Clipboard.put <sexp>	134
Encryption	134
Encrypt {<pw_sexp>}, <source_sexp>, <encrypted_svar>	134
Decrypt <pw_sexp>, <encrypted_sexp>, <decrypted_svar>	135
Ringer	135
Ringer.get.mode <nvar>	135

Ringer.set.mode <nexp>	135
Ringer.get.volume <vol_nvar> { , <max_nvar> }	135
Ringer.set.volume <nexp>	135
String Operations	136
Join <source_array\$[]>, <result_svar> {, <separator_sexp>{, <wrapper_sexp>}}	136
Join.all <source_array\$[]>, <result_svar> {, <separator_sexp>{, <wrapper_sexp>}}	136
Split <result_array\$[]>, <sexp> {, <test_sexp> }	136
Split.all <result_array\$[]>, <sexp> {, <test_sexp> }	136
Speech Conversion	137
Text To Speech	137
TTS.init	138
TTS.speak <sexp> {, <wait_lexp> }	138
TTS.speak.toFile <sexp> {, <path_sexp> }	138
TTS.stop	138
Speech To Text (Voice Recognition)	138
STT.listen {<prompt_sexp> }	138
STT.results <string_list_ptr_nexp>	139
Information About Your Android Device	140
Device Command Overview	140
Device <svar>	141
Device <nexp> <nvar>	141
Device.Language <svar>	141
Device. Locale <svar>	142
Phone.info <nexp> <nvar>	142
Screen.rotation, size[], realsize[], density	143
Screen.rotation <nvar>	144
Screen.size, size[], realsize[], density	144
WiFi.info {{<SSID_svar>},{, <BSSID_svar>},{, <MAC_svar>},{, <IP_var>},{, <speed_nvar>}}	144
Running in the Background	145
Home	145
OnBackground:	145
Background.resume	145

WakeLock <code_nexp>{, <flags_nexp>}	145
WifiLock <code_nexp>	146
Miscellaneous Commands	147
Headset <state_nvar>, <type_svar>, <mic_nvar>	147
Notify <title_sexp>, <subtitle_sexp>, <alert_sexp>, <wait_lexp>	147
Pause <ticks_nexp>	148
Swap <nvar_a> <sva_a>, <nvar_b> <sva_b>	148
Tone <frequency_nexp>, <duration_nexp> {, <duration_chk_lexp>}	148
Vibrate <pattern_array[{<start>,<length>}]>, <nexp>	148
VolKeys	149
VolKeys.off	149
VolKeys.on	149
SQLITE	149
Overview	149
SQLITE Commands	150
Sql.open <DB_pointer_nvar>, <DB_name_sexp>	150
Sql.close <DB_pointer_nvar>	150
Sql.new_table <DB_pointer_nvar>, <table_name_sexp>, C1\$, C2\$, ...,CN\$	150
Sql.drop_table <DB_pointer_nvar>, <table_name_sexp>	150
Sql.insert <DB_pointer_nvar>, <table_name_sexp>, C1\$, V1\$, C2\$, V2\$, ..., CN\$, VN\$	150
Sql.query <cursor_nvar>, <DB_pointer_nvar>, <table_name_sexp>, <columns_sexp> {, <where_sexp> {, <order_sexp> } }	151
Sql.query.length <length_nvar>, <cursor_nvar>	151
Sql.query.position <position_nvar>, <cursor_nvar>	151
Sql.next <done_lvar>, <cursor_nvar>{, <cv_svars>}	152
Sql.delete <DB_pointer_nvar>, <table_name_sexp>{,<where_sexp>,<count_nvar> } }	152
Sql.update <DB_pointer_nvar>, <table_name_sexp>, C1\$, V1\$, C2\$, V2\$,...,CN\$, VN\${: <where_sexp>}	153
Sql.exec <DB_pointer_nvar>, <command_sexp>	153
Sql.raw_query <cursor_nvar>, <DB_pointer_nvar>, <query_sexp>	153
Graphics	153
Introduction	153

The Graphics Screen and Graphics Mode	153
Display Lists	154
Drawing Coordinates	154
Drawing into Bitmaps.....	155
Colors	155
Paints.....	155
Style.....	156
Hardware-accelerated Graphics	157
Graphics Setup Commands.....	158
Gr.open {{alpha}}, red, green, blue, <ShowStatusBar_lexp>, <Orientation_nexp>}}	158
Gr.color {{alpha}}, red, green, blue, style, paint}}	158
Gr.set.antialias {{<lexp>}, <paint_nexp>}}	159
Gr.set.stroke {{<nexp>}, <paint_nexp>}}	159
Gr.orientation <nexp>.....	160
Gr.statusbar {<height_nvar> } {, showing_lvar}.....	160
Gr.statusbar.show <nexp>	160
Gr.render.....	160
Gr.screen width, height, density }	161
Gr.scale x_factor, y_factor	161
Gr.cls	161
Gr.close	162
Gr.front flag.....	162
Gr.brightness <nexp>.....	162
Graphical Object Creation Commands.....	162
Gr.point <obj_nvar>, x, y.....	162
Gr.line <obj_nvar>, x1, y1, x2, y2.....	163
Gr.rect <obj_nvar>, left, top, right, bottom.....	163
Gr.oval <obj_nvar>, left, top, right, bottom	163
Gr.arc <obj_nvar>, left, top, right, bottom, start_angle, sweep_angle, fill_mode.....	163
Gr.circle <obj_nvar>, x, y, radius.....	164
Gr.set.pixels <obj_nvar>, pixels[{{<start>}, <length>}}] {x, y}.....	164
Gr.poly <obj_nvar>, list_pointer {x, y}	164

Groups.....	165
Gr.group <object_number_nvar>{, <obj_nexp>}.....	166
Gr.group.list <object_number_nvar>, <list_ptr_nexp>	166
Gr.group.getDL <object_number_nvar>.....	166
Gr.group.newDL <object_number_nvar>	166
Hide and Show Commands	167
Gr.hide <object_number_nexp>.....	167
Gr.show <object_number_nexp>	167
Gr.show.toggle <object_number_nexp>	167
Touch Query Commands.....	167
Gr.touch touched, x, y.....	167
Gr.bounded.touch touched, left, top, right, bottom	168
Gr.touch2 touched, x, y.....	168
Gr.bounded.touch2 touched, left, top, right, bottom	168
OnGrTouch:	168
Gr.onGrTouch.resume	168
Text Commands	168
Overview	168
Gr.text.align {{<type_nexp>},{<paint_nexp>}}.....	169
Gr.text.bold {{<lexp>},{<paint_nexp>}}.....	169
Gr.text.size {{<size_nexp>},{<paint_nexp>}}.....	169
Gr.text.skew {{<skew_nexp>},{<paint_nexp>}}	169
Gr.text.strike {{<lexp>},{<paint_nexp>}}.....	170
Gr.text.underline {{<lexp>},{<paint_nexp>}}	170
Gr.text.setfont {{<font_ptr_nexp> <font_family_sexp>} {, <style_sexp>} {,<paint_nexp>}}.....	170
Gr.text.typeface {{<font_nexp>} {, <style_nexp>} {,<paint_nexp>}}	171
Gr.text.height {<height_nvar>} {, <up_nvar>} {, <down_nvar>}.....	171
Gr.text.width <nvar>, <exp>	172
Gr.get.textbounds <exp>, left, top, right, bottom	172
Gr.text.draw <object_number_nvar>, <x_nexp>, <y_nexp>, <text_object_sexp>	173
Bitmap Commands.....	174
Overview	174

Gr.bitmap.create <bitmap_ptr_nvar>, width, height	174
Gr.bitmap.load <bitmap_ptr_nvar>, <file_name_sexp>	174
Gr.bitmap.size <bitmap_ptr_nexp>, width, height.....	175
Gr.bitmap.scale <new_bitmap_ptr_nvar>, <bitmap_ptr_nexp>, width, height {, <smoothing_lexp>}	175
Gr.bitmap.delete <bitmap_ptr_nexp>.....	175
Gr.bitmap.crop <new_bitmap_ptr_nvar>, <source_bitmap_ptr_nexp>, <x_nexp>, <y_nexp>, <width_nexp>, <height_nexp>	175
Gr.bitmap.save <bitmap_ptr_nvar>, <filename_sexp>{, <quality_nexp>}.....	176
Gr.bitmap.draw <object_ptr_nvar>, <bitmap_ptr_nexp>, x , y	176
Gr.get.bmpixel <bitmap_ptr_nvar>, x, y, alpha, red, green, blue	176
Gr.bitmap.fill <bitmap_ptr_nexp>, <x_nexp>, <y_nexp>	176
Gr.bitmap.drawinto.start <bitmap_ptr_nexp>.....	176
Gr.bitmap.drawinto.end	177
Paint Commands.....	177
Gr.paint.copy {{<src_nexp>}{, <dst_nexp>}}.....	177
Gr.paint.get <object_ptr_nvar>	177
Gr.paint.reset {<nexp>}.....	178
Rotate Commands	178
Gr.rotate.start angle, x, y{<obj_nvar>}.....	178
Gr.rotate.end {<obj_nvar>}.....	178
Camera Commands.....	178
Gr.camera.select 1 2	179
Gr.camera.shoot <bm_ptr_nvar>	179
Gr.camera.autoshoot <bm_ptr_nvar>{, <flash_mode_nexp> {, focus_mode_nexp} }	180
Gr.camera.manualShoot <bm_ptr_nvar>{, <flash_mode_nexp> {, focus_mode_nexp} }	180
Other Graphics Commands.....	180
Gr.screen.to_bitmap <bm_ptr_nvar>.....	180
Gr.get.pixel x, y, alpha, red, green, blue	181
Gr.save <filename_sexp> {,<quality_nexp>}.....	181
Gr.get.type <object_ptr_nexp>, <type_svar>.....	181
Gr.get.params <object_ptr_nexp>, <param_array\$[]>	181

Gr.get.position <object_ptr_nexp>, x, y	181
Gr.move <object_ptr_nexp> {{, dx}{, dy}}	181
Gr.get.value <object_ptr_nexp> {, <tag_sexp>, <value_nvar value_svar>}...	182
Gr.modify <object_ptr_nexp> {, <tag_sexp>, <value_nexp value_sexp>}	182
GR_COLLISION(<object_1_nexp>, <object_2_nexp>)	184
Gr.clip <object_ptr_nexp>, <left_nexp>, <top_nexp>, <right_nexp>, <bottom_nexp>{, <RO_nexp>}	184
Gr.newDL <dl_array[{{<start>,<length>}}]>	185
Gr.getDL <dl_array[]> {, <keep_all_objects_lexp> }	185
Audio Interface	186
Introduction	186
The Audio Interface	186
Audio File Types	186
Commands	186
Audio.load <aft_nvar>, <filename_sexp>	186
Audio.play <aft_nexp>	186
Audio.stop	187
Audio.pause	187
Audio.loop	187
Audio.volume <left_nexp>, <right_nexp>	187
Audio.position.current <nvar>	187
Audio.position.seek <nexp>	188
Audio.length <length_nvar>, <aft_nexp>	188
Audio.release <aft_nexp>	188
Audio.isdone <lvar>	188
Audio.record.start <fn_svar>	188
Audio.record.stop	188
SoundPool	188
Introduction	188
Commands	189
Soundpool.open <MaxStreams_nexp>	189
Soundpool.load <soundID_nvar>, <file_path_sexp>	189

Soundpool.unload <soundID_nexp>.....	189
Soundpool.play <streamID_nvar>, <soundID_nexp>, <rightVolume_nexp>, <leftVolume_nexp>, <priority_nexp>, <loop_nexp>, <rate_nexp>	189
Soundpool.setvolume <streamID_nexp>, <leftVolume_nexp>, <rightVolume_nexp>	190
Soundpool.setrate <streamID_nexp>, <rate_nexp>.....	190
Soundpool.setpriority <streamID_nexp>, <priority_nexp>	190
Soundpool.pause <streamID_nexp>	190
Soundpool.resume <streamID_nexp>	190
Soundpool.stop <streamID_nexp>	190
Soundpool.release	190
GPS	190
GPS Control commands	191
Gps.open {{<status_nvar>},{<time_nexp>},{<distance_nexp>}}	191
Gps.close	191
Gps.status {{<status_var>}, {<infix_nvar>},{inview_nvar}, {<sat_list_nexp>}}	191
GPS Location commands.....	193
Gps.location {{<time_nvar>}, {<prov_svar>}, {<count_nvar>}, {<acc_nvar>}, {<lat_nvar>}, {<long_nvar>}, {<alt_nvar>}, {<bear_nvar>}, {<speed_nvar>}}.....	194
Gps.time <nvar>	194
Gps.provider <svar>	194
Gps.satellites {{<count_nvar>}, {<sat_list_nexp>}}.....	195
Gps.accuracy <nvar>	195
Gps.latitude <nvar>.....	195
Gps.longitude <nvar>.....	195
Gps.altitude <nvar>.....	195
Gps.bearing <nvar>	195
Gps.speed <nvar>.....	195
Sensors	195
Introduction	195
Sensor Commands	196
Sensors.list <sensor_array\$[]>	196
Sensors.open <type_nexp>{:<delay_nexp>},{<type_nexp>{:<delay_nexp>}, ...}	196

Sensors.read sensor_type_nexp, p1_nvar, p2_nvar, p3_nvar	197
Sensors.close.....	197
System.....	197
System Commands.....	198
System.open.....	198
System.write <sexp>	198
System.read.ready <nvar>	198
System.read.line <svar>	198
System.close.....	198
Superuser Commands.....	198
Su.open	198
Su.write <sexp>	198
Su.read.ready <nvar>	198
Su.read.line <svar>.....	198
Su.close	199
App Commands.....	199
App.broadcast <action_sexp>, <data_uri_sexp>, <package_sexp>, <component_sexp>, <mime_type_sexp>, <categories_sexp>, <extras_bptr_nexp>, <flags_nexp>	199
App.start <action_sexp>, <data_uri_sexp>, <package_sexp>, <component_sexp>, <mime_type_sexp>, <categories_sexp>, <extras_bptr_nexp>, <flags_nexp>	199
Appendix A – Command List	201
Appendix B – Sample Programs	213
Appendix C – Launcher Shortcut Tutorial	214
Introduction	214
How to Make a Shortcut Application (older versions of Android—prior to Android 4.0)	214
How to Make a Shortcut Application (newer versions of Android—Android 4.0 and later)	215
What you need to know	216
Appendix D – Building a Standalone Application.....	217
Introduction	217
License Information	217
Before You Start	217
Setting Up the Development Environment.....	217

Download the BASIC! Source Code from the GitHub Repository	218
Download the BASIC! Source Code from the Legacy Archive	218
Create a New Project in Eclipse	219
Rename the Package.....	220
In Eclipse	220
Other IDEs	221
Renaming complete	222
Modifications to setup.xml	222
Advanced Customization with setup.xml	223
Files and Resources.....	225
Testing the APK	226
Installing a BASIC! Program into the Application.....	227
Application Icons.....	228
Modifications to the AndroidManifest.xml File.....	229
Setting the Version Number and Version Name.....	230
Permissions	230
Disable the Shortcut Launcher.....	230
Launch at device boot.....	231
Preferences	232
Finished	233
Appendix E – BASIC! Distribution License	234
Apache Commons	245

Changes in this Version

- Clarifications and additional explanations:
 - Labels may not start with keywords. Variable names may, but this is not recommended.
 - Clarified precedence of logical and arithmetic operators.
 - A file may be **Included** only once in a program.
 - OnMenuKey: does not work on many devices, because they have no MENU key.
 - More complete explanation of how BASIC! paths differ from Android paths.
 - Better explanation of **Array.Copy**.
 - Copied description of file modes from **Byte.open** to **Text.open**.
 - Clarification of what is copied by **Byte.Copy**.
 - In **Appendix D**, described file-handling commands that cannot read **res** or **assets**.
- Corrected errors:
 - The "\n" metacharacter is not CR/LF, it is just LF.
 - Fixed code sample for computed **Goto**.
- New BASIC! functions and commands:
 - **Device.language** and **Device.locale**
 - **Program.info**
 - Six new **ZIP** commands: **count**, **dir**, **open**, **close**, **read**, **write**.
- Other changes in BASIC!:
 - Expanded the **About** option of the Editor Menu. Includes a link to BASIC!'s Privacy Policy as required for distribution on Google Play.
 - Added "ENCRYPT_RAW" and "DECRYPT_RAW" types to **ENCODE\$()** and **DECODE\$()**.
 - Most **ENCODE\$()**, **DECODE\$()**, **ENCRYPT**, and **DECRYPT** errors do not stop your program. Instead, they write an error message you can read with **GETERROR\$()**.
 - Added flags to **Wake.lock** to control screen behavior when lock is acquired or released.
 - **Audio.load** can take a URL as well as a file name.

About the Title, De Re BASIC!

"De Re" is Latin for "of the thing" or "about".

About the Cover Art

Thanks to BASIC! collaborator Nicolas Mougin. The images are screenshots from real BASIC! programs available from the Google Play™ store, or from the excellent collection of shared BASIC! programs available at <http://laughton.com/basic/programs>.

You can find more information on the BASIC! user forum at <http://rfobasic.freeforums.org/shared-basic-programs-f6.html>.

Credits

Thanks to Paul Laughton, the original creator of BASIC! and of this document. The first edition was published in 2011. Mr. Laughton placed this document in the Public Domain in 2016.

Thanks also to Mike Leavitt of Lansdowne, VA, USA, for his many contributions and long-time support.

Technical Editor

The current editor of this manual monitors <https://github.com/RFO-BASIC/De-Re-Basic/issues> and the BASIC! user forum at <http://rfobasic.freeforums.org/suggestions-for-improving-the-manual-f9.html> for corrections and suggestions.

Getting BASIC!

You can get BASIC! for your Android device from the **Google Play™** store.

If you prefer to download the installation file (Basic.apk) yourself, or to get a previous version, get it from **Bintray** at <https://bintray.com/rfo-basic/android/RFO-BASIC/>

This manual is also available from Bintray, in PDF format. BASIC! collaborator Nicolas Mougin keeps the manual in HTML format on his BASIC! website: <http://rfo-basic.com/manual>. The document source is available from GitHub at <https://github.com/RFO-BASIC/De-Re-Basic>.

BASIC! Forum

Join the community of BASIC! users at <http://rfobasic.freeforums.org>, where you are always welcome.

BASIC! Tutorial

A BASIC! user, Nick Antonaccio, has written a very nice tutorial for BASIC! You can find it at <http://rfobasic.com>.

BASIC! Operation

Permissions

This application requests many permissions, permissions such as sending and receiving SMS messages, making phone calls, record audio, etc. BASIC! does not exercise any of these permissions (except writing to the SD card) on its own. These permissions get exercised by the BASIC! programmer, you. You and only you. You exercise these permissions by means of the programs that you write.

If you write a program that uses the `sms.send` command then BASIC! will attempt to send an SMS message. BASIC! must have permission to send SMS messages for this command to work. If you never use the `sms.send` command then BASIC! will never send an SMS message. You are in control.

The source code for BASIC! is available from the BASIC! web site (<http://laughton.com/basic/>) and the BASIC! GitHub repository (<https://github.com/RFO-BASIC/Basic>). Please feel free to examine this source code if you have any doubt about the use of these permissions.

Editor

Editing the Program

The Editor is where BASIC! programs are written and edited. The operation of the Editor is fairly simple. Tap the screen at the point where you want to edit the program. A cursor will appear. Use the keyboard to edit at the cursor location.

When the Enter key is tapped, the new line will automatically indent to the indent level of the previous line. This feature will not work if the Preference, "Editor AutoIndent," is not checked. This feature also may not work if you are using a software keyboard.

If the program that you are editing has been given a name via Save or Load then that program name will be shown in the title bar.

Some Android devices are shipped with "Settings/Developer Option/Destroy Activities" checked and/or "Settings/Energy/Quick Restart" checked. Both of these setting create problems with loading files into the Editor. It appears as if you have gone through the process of loading the file but nothing appears in the editor. The solution to the problem is to uncheck both of these options. Even better, completely turn off Developer Options unless you know that you have a legitimate development need.

If your Android device does not have a physical keyboard, you will see a virtual keyboard. If you see the virtual keyboard, then you will see different things depending upon the way you are holding the device. If the device is in landscape mode then you will see a dialog box with a chunk of the program in a small text input area. You can scroll the small chunk of text up and down in this area but you will not be able to see very much of the program at any one time. It is probably best not to try to edit a program in landscape mode; hold your device in portrait mode while editing.

On some devices, if you do a long touch on the screen, a dialog box will appear. You can use the selections in the box for selecting, copying, cutting and pasting of text, among other things. Other devices have different procedures for invoking the cut and paste functions.

Multiple Commands on a Line

More than one BASIC! source code statement may be written on one physical line. Separate commands with a colon character ":".

For example, the following line uses three separate commands to initialize some variables:

```
name$="BASIC!" : ver=1.86 : array.load reviews$[], "Great!", "Wow!", "Fantastic!"
```

Note: two commands, **Sensors.open** and **SQL.update**, use the colon as a sub-parameter separator. If you use multiple-command lines, be careful when using these two commands.

Line Continuation

A BASIC! source code statement may be written on more than one physical line using the line continuation character "~". If "~" is the last thing on a line, except for optional spaces, tabs, or a '%' comment, the line will be merged with the next line. This behavior is slightly different in the **Array.load** and **List.add** commands; see the descriptions of those commands for details.

Note: this operation is implemented by a preprocessor that merges the source code lines with continuation characters before the source code is executed. If you have a syntax error in the merged line, it will show as one line in the error message, but it will still be multiple lines in the editor. Only the first physical line will be highlighted, regardless of which line the error is in.

For example, the code line:

```
s$ = "The quick brown fox " + verb$ + " over " + count$ + " lazy dogs"
```

could be written as:

```
s$ = "The quick brown fox " +~  
    verb$ +      ~      % what the fox did  
    " over " +  ~  
    count$ +   ~      % how many lazy dogs  
    " lazy dogs"
```

- Format Line

If a line has the # character at the end of the line, the keywords in that line will be capitalized and the # will be removed.

This feature may not work if you are using a virtual keyboard.

This feature will not work if the Preference option "Editor AutoIndent" is not checked.

Menus

Press the MENU key or touch the Menu icon to access the following menus. On some versions of Android, you will not see all of the menu options. Instead, you will see the first five options and a **More** options. Select the **More** option to see all of the options listed.

Run

Run the current program.

If the program has been changed since it was last saved, you will be given an opportunity to save the program before the run is started.

If a run-time error occurs then the offending line will be shown as selected in the editor.

Load

Load a program file into the editor. The first time you open BASIC! after installing it, if you select **Load** it displays the sample programs in the directory **rfo-basic/source/Sample_Programs**. (See **Paths Explained**, later in this manual.) Otherwise, when you open BASIC! and select **Load** it starts in the default source directory **rfo-basic/source**. Program files must have the extension **.bas**.

BASIC! checks to see if the current program in the Editor has been changed when **Load** is tapped. You will be offered the opportunity to save the program if it has been changed. If you choose to save the program, **Load** will restart after the save is done.

The "BASIC! Load File" screen shows the path to your current directory followed by sorted lists of the subdirectories and program files in it. Directories are denoted by the **(d)** appended to the name. BASIC! programs are shown with the **.bas** extension. If there are files in the directory that do not have the **.bas** extension, they do not appear in the list.

Tap on a **.bas** file to load it into the Editor.

You can navigate to any directory on your device for which you have **read** permission.

- Tap on a directory to display its contents.
- Tap on the **".."** at the top of list to move up one directory level. The tap has no effect if the current directory is the device root directory **"/"**.

You can exit the **Load** option without loading a program by tapping the BACK key.

BASIC! remembers the path to the directory you are in when you load a program. Next time you select **Load**, it starts in that directory. If you select **Save** or **Save and Run**, the file is saved in the remembered directory (unless it is **Sample_Programs**).

Save

Saves the program currently in the editor.

A dialog box displays the path to the directory where you will save the file and an input area where you can enter the file name. If the current program has a name because it was previously loaded or saved then that name will be in the text input area. Type in the name you want the file saved as and tap **OK**. The extension **.bas** will be added to file name if it is not already there.

If you do not enter a file name, BASIC! uses a default filename, **default.bas**.

BASIC! remembers the path to the directory you were in when you last loaded or saved a program. When you **Save**, the file name you type is saved in the remembered directory. If the name you type includes subdirectories, BASIC! remembers the new path. The name you type can include "../". Be careful if you are using a soft keyboard, as it may automatically insert spaces that you don't want.

You cannot save programs in the sample program directory **source/Sample_Programs**. If you load a program from **source/Sample_Programs**, change it, and **Save** it, the program is saved in **source**.

You can exit **Save** without saving a file by tapping the BACK key.

Clear

Clear the current program in the Editor. You will be offered the opportunity to save the current program if it has been changed.

Search

Search for strings in the program being edited. Found strings may be replaced with a different string.

The Search view shows a Text Window with the text from the Editor, a **Search For** field and a **Replace With** field.

If there is a block of text currently selected in the Editor, then that text will be placed into the **Search For** field.

The initial location of the search cursor will be at the start of the text regardless of where the cursor was in the Editor text.

Note: The search ignores case. For example, searching for "basic" will find "BASIC" This is because BASIC! converts the whole program to lower case (except characters within quotes) when the program is run.

NEXT BUTTON

Start the search for the string in the **Search For** field. The search is started at the current cursor location. If the string is found then it will be selected in **Text Window**.

If the **Done** button is tapped at this point then the Editor will returned to with the found text selected.

If the **Replace** button is tapped then the selected text will be replaced.

Pressing the **Next** button again will start a new search starting at the end of the selected or replaced text.

If no matching text is found then a "string not found" message is shown. Tapping the **Done** button returns to the Editor with the cursor at the end of the program. Alternatively, you could change the **Search For** text and start a new search.

REPLACE BUTTON

If **Next** has found and selected some text then that text is replaced by the contents of the **Replace With** field.

If no text has been found then the message, "Nothing found to replace" will be shown.

REPLACE ALL BUTTON

All occurrences of the **Search For** text are replaced with the **Replace With** text. **Replace All** always starts at the start of the text. The last replaced item will be shown selected in the **Text Window**. The number of items replaced will be shown in a message.

DONE BUTTON

Returns to the Editor with the changed text. If there is selected text in the **Text Window** then that text will be shown selected in the Editor.

BACK KEY

Returns to the Editor with the original text unchanged. All changes made during the Search will be undone. Think of the BACK key as UNDO ALL.

Load and Run

Selecting this option is exactly the same as first selecting **Load** and then selecting **Run**. The selected program is loaded into the Editor and is run immediately.

Save and Run

Selecting this option is a fast way to save and then run. Any changes you have made are saved, overwriting your file, and your program is run immediately. A brief popup notifies you that your file has been changed. If the program you are editing has no name (not previously loaded or saved), the Editor will ask you what name to use.

Format

Format the program currently in the Editor. The keywords are capitalized. Program lines are indented as appropriate for the program structure. Left- and right-double quotation marks (" and ") are replaced by simple ASCII quotation marks (").

When copying program text from the Forum or another web site, "non-breaking space" characters, designated ** ** in HTML, may be inserted into the program text. Except when they are enclosed in quoted strings, **Format** converts these characters to simple ASCII spaces.

Delete

Delete files and directories. The command should be used for maintaining files and directories that are used in BASIC! but it can also be used to delete any file or directory on the SD card for which you have the required permissions. **Delete** starts in the **rfo-basic** directory.

Tapping **Delete** presents the "BASIC! Delete File" screen. The screen shows the path to your current directory followed by sorted lists of the directories and files in it. Directories are marked with **(d)** appended to the name and appear at the top of the list.

Tapping a file name displays the "Confirm Delete" dialog box. Tap the **Delete** button to delete the file. Tap the **No** button to dismiss the dialog box and not delete the file.

Tapping a directory name displays the contents of the directory. If the directory is empty the "Confirm Delete" dialog box is shown. Tap the **Delete** button to delete the directory. Tap the **No** button to dismiss the dialog box and not delete the directory.

Tap the **..** at the top of the screen to move up one directory level. Tapping the **..** has no effect if you are in the root directory **/**.

Exit **Delete** by tapping the BACK key.

Preferences

SCREEN COLORS

Opens a sub-menu with options for setting the colors of the various screens in BASIC!

COLOR SCHEME

Sets the color scheme of the screens. The schemes are identified by their appearance with the default colors. Choose Black text on a White background, White text on a Black background or White text on a Blue background.

CUSTOM COLORS

Check the box to override the Color Scheme setting, allowing you to set your own colors. You can set the Text (foreground) Color, the Background Color, the Line Color, and the Highlight Color. Each color is specified as a single number with 16 hexadecimal digits: four fields of four digits each for Alpha (opacity), Red, Green, and Blue components.

CONSOLE SETTINGS

Opens a sub-menu with options for settings of the Console and various others screens in BASIC!

FONT SIZE

Sets the font size (Small, Medium, or Large) to be used with the various screens in BASIC!.

TYPEFACE

Choose the typeface to be used on the Output Console and some other screens: Monospace, Sans Serif, or Serif.

CONSOLE MENU

Check the box if the Menu should be visible in the Output Console and TGet screen.

CONSOLE LINES

Check the box if the text lines in the Output Console should be underlined.

EMPTY CONSOLE COLOR

Choose the background color of the part of the Output Console that has not yet been written. It can match the background color of the text or the color of the lines separating text lines.

This setting also applies to the **Select** (but not **Dialog.select**) command.

EDITOR SETTINGS

Opens a sub-menu with options for setting properties and features of the Program Editor.

EDITOR LINES

Check the box if the text lines in the Editor should be underlined.

EDITOR LINE WRAP

Check the box if long text lines in the Editor should wrap at the edge of the screen. If unchecked, long lines are not wrapped, and the Editor screen may be scrolled horizontally.

EDITOR AUTOINDENT

Check the box if you want the Editor to do auto indentation. Enabling auto indentation also enables the formatting of a line that ends with the "#" character.

Some devices are not able to do auto indenting properly. In some of those devices the AutoIndent feature may cause the Editor to be unusable. If that happens, turn off AutoIndent.

MENU ITEMS ON ACTION BAR

Opens a sub-menu with options for moving some of the Editor menu items to the Action Bar, if there is room for them there. You can select as many as you like, but the number of items moved depends on the device and orientation. These options have no effect on Android devices before Honeycomb (3.0).

RUN ON ACTION BAR

If checked, the Editor will attempt to move the RUN item from the Menu to the Action Bar.

LOAD ON ACTION BAR

If checked, the Editor will attempt to move the LOAD item from the Menu to the Action Bar.

SAVE ON ACTION BAR

If checked, the Editor will attempt to move the SAVE item from the Menu to the Action Bar.

EXIT ON ACTION BAR

If checked, the Editor will attempt to move the EXIT item from the Menu to the Action Bar.

SCREEN ORIENTATION

Choose to allow the Sensors to determine the orientation of the screens or to set a fixed orientation without regard to the Sensors.

Note: The reverse orientations apply to Android 2.3 or newer.

GRAPHIC ACCELERATION

Check this box to enable GPU-assisted graphics acceleration on devices since 3.0 (Honeycomb) that support it. It is disabled by default. If you enable this option, test your program carefully. Hardware acceleration can make some of BASIC!'s graphical operations fail.

BASE DRIVE

Some Android devices have several external storage devices (and some have no physical external storage devices). BASIC! will use the system-suggested device as its base drive. The *base drive* is the device where the BASIC! "rfo-basic" directory (base directory) is located. The *base directory* is where BASIC!'s programs and data are stored. (See "Working with Files", later in this manual.)

If your device does have more than one external storage device they will be listed here. If your device has no external storage devices, your one and only choice will be "No external storage". Tap the device you want to use as the base drive and press the BACK key. You will then be given the choice of either immediately restarting BASIC! with the new base drive or waiting and doing the restart yourself.

In this manual, <pref base drive> means the base drive you selected when you set the Base Drive here.

Note: If you have created a Launcher Shortcut (see Appendix C) with files in one base directory but try to execute that shortcut while using a different base directory, the shortcut will fail to execute.

Commands

The Commands command presents the list of the BASIC! commands and functions as copied from Appendix A of this document.

Tapping an alpha key will cause the command list to scroll to commands that start with that character. There will be no scrolling if there is no command that starts with that character.

Note: You can hide the virtual keyboard with the BACK key. If you do that, you will not be able to get it back until you invoke the **Commands** option again.

Tapping on a particular command causes that command to be copied to the clipboard (not including the page number) and returning to the Editor. You can then paste the command into your BASIC! program.

About

The **About** option displays the version of BASIC! that you are using, followed by a set of buttons that connect you to various websites with information about BASIC!. Make sure that you have a connection to the Internet before selecting one of the **About** buttons.

BASIC! HOME PAGE

Opens the BASIC! home page, <http://rfo-basic.com>.

BASIC! USER FORUM

Opens the BASIC! user forum site, <http://rfobasic.freeforums.org>.

BASIC! USERS' PROGRAMS

Opens a repository of BASIC! programs written by members of the BASIC! user forum, hosted by the founder of BASIC, Paul Laughton. The programs are found at <http://laughton.com/basic/programs>.

BASIC! ON BINTRAY

Opens the BASIC! page of the binary hosting site Bintray, <https://bintray.com/rfo-basic/android/RFO-BASIC>. At this site, you can download this manual in PDF format and the Android installable (.apk file). Both are available for recent versions of BASIC!. You can also read the release notes for each version.

BASIC! ON GITHUB

Opens the BASIC! page of the software development site GitHub, <https://github.com/RFO-BASIC>. At this site you can download the source for every version of BASIC! since v01.69. You can also download the "source" for this manual, as a Microsoft™ Word™ document, for each version.

BASIC! OPEN-SOURCE LICENSE

Displays the GPLv3 license under which BASIC! is distributed. This copy of the license is hosted at <http://laughton.com/basic/license.html>. This license is reproduced in Appendix E of this manual.

BASIC! PRIVACY POLICY

Displays the BASIC! Privacy Policy. Installing BASIC! requires several Android permissions, some of which Google considers sensitive. The BASIC! program interpreter does not use any of these permissions. Except for limited use by the Editor, these permissions are requested only to enable your BASIC! programs to use the full power of the Android platform. This is explained in the Privacy Policy, which is hosted at <http://rfo-basic.com/PrivacyPolicy.html>.

Exit

The only way to cleanly exit BASIC! is to use the **Exit** option.

Pressing the HOME key while in BASIC! leaves BASIC! in exactly the same state it was in when the HOME key was tapped. If a program was running, it will still be running when BASIC! is re-entered. If you were in the process of deleting, the Delete screen will be shown when BASIC! is re-entered.

Run

Selecting **Run** from the Editor's menu starts the program running. However, if the source in the Editor has been changed, then the Save dialog will be displayed. You may choose to save the changed source or continue without saving.

The BASIC! Output Console will be presented as soon as the program starts to run. You will not see anything on this screen unless one of the following situations occur:

- the program prints something
- the **END** statement is executed
- you are in Echo mode
- there is a run-time error

If the program does not print anything then the only indication you would get that the program has finished is if the program ends with an **End** statement.

If the program does not contain any executable statements then the message, "Nothing to execute" will be displayed.

Tapping the BACK key will stop a running program. Tapping the BACK key when the program run has ended will restart the Editor.

If the program ended with a run-time error, the line where the error occurred will be shown selected in the Editor. If the error occurred in an **INCLUDE** file then the **INCLUDE** statement will be shown selected.

The Editor cursor will remain where it was when the **Run** was started if no run-time error occurred.

Menu

Pressing the MENU key or touching the Menu icon while a program is running or after the program is stopped will cause the Run Menu to be displayed. (Except when Graphics is running. See the Graphics section for details.)

Stop

If a program is running, the **Stop** menu item will be enabled. Tapping **Stop** will stop the running program. **Stop** will not be enabled if a program is not running.

Editor

Editor will not be enabled if a program is running. If the program has stopped and **Editor** is thus enabled then selecting **Editor** will cause the Editor to be re-entered. You could also use the BACK key to do this.

Crashes

BASIC! is a very large and complex program that is constantly under development. From time to time, it will crash. The best way to report a crash is to visit the BASIC! forum.

A BASIC! Program

A BASIC! program is made up of lines of text. With a few exceptions that will be explained later, each line of text is one or more **statements**. If a line has more than one statement they are separated by colon (":") characters.

A statement always consists of a single command, usually followed by one or more parameters that are separated by commas. Here is a simple BASIC! program:

```
PRINT "Hello, World!"
```

This program has one statement. The command is **PRINT**. It has one parameter, the string constant **"Hello, World!"**. (A string constant, or string literal, is a set of characters enclosed in double quotation marks. This, too, will be explained later.)

If you start the BASIC! app, so you are in the Editor, you can type in this one-line program. Then you can select **Run** from the Editor's menu. BASIC! will run your program. When the program is done running, you see the **Console**, BASIC!'s, output screen, with **Hello, World!** printed at the top.

Command Description Syntax

Upper and Lower Case

Commands are described using both upper and lower case for ease of reading. BASIC! converts every character (except those between double quotation marks) to lower case when the program is run.

<nexp>, <sexp> and <lexp>

These notations denote a numeric expression (<nexp>), a string expression (<sexp>), and a logical expression (<lexp>). An expression can be a variable, a number, a quoted string or a full expression such as $(a \cdot x^2 + bx + c)$.

<nvar>, <svar> and <lvar>

This notation is used when a variable, not an expression, must be used in the command. Arrays with indices (such as $n[1,2]$ or $s[3,4]$) are considered to be the same as <nvar>, <svar> and <lvar>.

Array[] and Array\$[]

This notation implies that an array name without indices must be used.

Array[{<start>,<length>}] and Array\$[{<start>,<length>}]

In most contexts, numeric expressions inside the brackets are indices specifying a single array element. In some commands, a pair of numeric expressions specifies a segment of the array. Both the start index and length are numeric expressions, and both are optional. This notation is shorthand for:

```
Array [ { {<start_nexp>} {, <length_nexp>} } ]  
Array$ [ { {<start_nexp>} {, <length_nexp>} } ]
```

{something}

Indicates something optional.

{ A | B | C }

This notation suggests that a choice of either A, B, or C, must be made. For example:

```
Text.open {r|w|a}, fn...
```

Indicate that either "r" or "w" or "a" must be chosen:

```
Text.open r, fn...
Text.open w, fn...
Text.open a, fn...
```

X, ...

Indicates a variable-sized list of items separated by commas. At least one item is required.

{n} ...

Indicates an optional list with zero or more items separated by commas.

<statement>

Indicate an executable BASIC! statement. A <statement> is usually a line of code but may occur within other commands such as: **IF** <lexp> **THEN** <statement>.

Optional Parameters

Many statements have optional parameters. If an optional parameter is omitted, the statement assumes a default value or performs a default action.

If an optional parameter is omitted, use a comma to mark its place, so following parameters are handled correctly. However, if there are no following parameters, omit the comma, too. With a few special exceptions (like **Print**), no statement can end with a comma.

Numbers

You can type decimal numbers in a BASIC! program:

- A leading sign ("+" or "-"), a decimal point ("." only), and an exponent (power of 10) are optional.
- An exponent is "e" or "E" followed by a number. The number may have a sign but no decimal point.

If you use a decimal point, it **MUST** follow a digit. 0.15 is a valid number, but .15 is a syntax error.

Numbers in BASIC! are double-precision floating point (64-bit IEEE 754). This means:

- A printed number will always have decimal point. For example, 99 will print as "99.0". You can print numbers without decimal points by using the INT\$() or FORMAT\$() functions. For example, either INT\$(99) or FORMAT\$("###", 99) will print "99".
- A number with more than 7 significant digits will be printed in floating point format. For example, the number 12345678 will be printed as 1.2345678E7. INT\$() or FORMAT\$() can be used to print large numbers in other than floating point format.
- Mathematical operations on decimal values are imprecise. If you are working with currency you should multiply the number by 100 until you need to print it out. When you print it, divide by 100.

A logical value (false = 0, true <> 0) is a kind of number.

You can use string functions to convert numbers to strings. STR\$(), INT\$(), HEX\$() and a few others do simple conversions. FORMAT\$() and USING\$() can do more complex formatting.

For the purposes of this documentation, numbers that appear in a BASIC! program are called Numeric Constants.

Strings

Strings in BASIC! are written as any set of characters enclosed in quote (") characters. The quote characters are not part of the string. For example, **"This is a string"** is a string of 16 characters.

To include the quote character in a string, you must escape it with a backslash: \". For example:

```
Print "His name is \"Jimbo\" Jim Giudice."
```

prints: His name is "Jimbo" Jim Giudice.

Newline characters may be inserted into a string with the escape sequence \n:

```
Print "Jim\nGiudice"
```

prints:

```
Jim
Giudice
```

"\n" can mean different things on different systems. In BASIC!, it is the same as an ASCII LF (line feed) character.

You can use another escape sequence, \t, to put a TAB character into a string. To embed a backslash, escape it with another backslash: \\. Other special characters can be inserted using the **CHR\$()** function.

Strings with numerical characters can be converted to BASIC! numbers using the **VAL(<sexp>)** function.

For the purposes of this documentation, strings that appear within a BASIC! program are called String Constants.

Variables

A BASIC! variable is a container for some numeric or string value.

Variable Names

Variable names must start with the characters "a" through "z", "#", "@", or "_". The remaining characters in the variable name may also include the digits, "0" through "9".

A variable name may be as long as needed.

Upper case characters can be used in variable names but they will be converted to lower case characters when the program is run. The variable name "gLoP" is the same as the name "glop" to BASIC!

You should avoid using variable names that start with BASIC! command keywords. Such variables are valid under most conditions, as will be explained later in this manual, but their use may cause confusion or errors. For example, **Donut = 5** is interpreted as **Do Nut=5**. BASIC! thus expects this **Do** statement to be followed by an **Until** statement somewhere before the program ends. A list of BASIC! commands can be found in **Appendix A**. See also the **Let** command and the **Expressions → Parentheses** section.

BASIC! statement labels and the names of user-defined functions, both described later in this manual, follow the same naming rules as BASIC! variables.

Variable Types

There are two types of variables: Variables that hold numbers and variables that hold strings. Variables that hold strings end with the character "\$". Variables that hold numbers do not end in "\$".

Age, **amount** and **height** are all numeric variable names.

First_Name\$, **Street\$** and **A\$** are all string variable names.

If you use a numeric variable without assigning it a value, it has the value 0.0. If you use a string variable without assigning it a value, its value is the empty string, "".

Scalar and Array Variables

There are two classes of variables: Scalars and Arrays.

Scalars

A scalar is a variable that can hold only one value. When a scalar is created it is assigned a default value. Numeric scalars are initialized to 0.0. String scalars are initialized to an empty, zero-length string, "".

You create a scalar variable just by using its name. You do not need to predeclare scalars.

Arrays

An array is variable that can hold many values organized in a systematically arranged way. The simplest array is the linear array. It can be thought of as a list of values. The array **A[index]** is a linear array. It can hold values that can accessed as **A[1]**, **A[2]**, ..., **A[n]**. The number (variable or constant) inside the square brackets is called the index.

If you wanted to keep a list of ten animals, you could use an array called **Animals\$[]** that can be accessed with an index of 1 to 10. For example: **Animals\$[5] = "Cat"**

Arrays can have more than one index or dimension. An array with two dimensions can be thought of as a list of lists. Let's assume that we wanted to assign a list of three traits to every animal in the list of animals. Such a list for a "Cat" might be "Purrs", "Has four legs" and "Has Tail". We could set up the

Traits array to have two dimensions such that Traits\$(5,2) = "Has four legs". If someone asked what are the traits of cat, search Animals\$(index) until "Cat" is found at index =5. Index=5 can then be used to access Traits(index,[1|2|3]).

BASIC! arrays can have any number of dimensions of any size.

BASIC! arrays are "one-based". This means that the first element of an array has an index of "1". Attempting to access an array with an index of "0" (or less than 0) will generate a run-time error.

Before an array can be used, it must be dimensioned using the **DIM** command. The **DIM** command tells BASIC! how many indices are going to be used and the sizes of the indices. Some BASIC! Commands automatically create a one-dimensional array. Auto-dimensioned array details will be seen in the description of those commands.

Note: It is recommended that the List commands (see below) be used in place of one-dimensional arrays. The List commands provide more versatility than the Array commands.

Array Segments

Some BASIC! Commands take an array as an input parameter. If the array is specified with nothing in the brackets (for example, "Animals\$[]"), then the command reads the entire array.

Most of these commands allow you to limit their operation to a segment of the array, using the notation "Array[start, length]", where both "start" and "length" are numeric expressions.

For example, you can write "Animals\$[2,3]". Usually that means "the animal at row 2 and column 3 of a two dimensional array called Animals\$". When used to specify an array segment, it has a different meaning: "read only the segment of the Animals\$ array that starts at index 2 and includes 3 items". Notice that this notation applies only to one-dimensional arrays. In fact, it treats all arrays as one-dimensional, regardless of how they are declared.

Both of the expressions in the "[start, length]" pair are optional. If the "start" value is omitted, the default starting index is 1. If the "length" value is omitted, the default is the length from the starting index to the end of the array. If both are omitted, the default is to use the entire array.

Array Commands

These commands all operate on Arrays. Commands operate on both numeric and string arrays, unless otherwise indicated.

Dim Array[<nexp>{, <nexp> } ...] { Array[<nexp>{, <nexp> } ...] } ...

The **Dim** command tells BASIC! how many dimensions an array will have and how big those dimensions are. BASIC! creates the array, reserving and initializing memory for the array data. All elements of a numeric array are initialized to the value 0.0. String array elements are initialized to the empty string, "". If you **Dim** an array that already exists, the existing array is destroyed and a new one created.

Multiple arrays can be dimensioned with one **Dim** statement. String and numeric arrays can be dimensioned in a single **Dim** command.

Examples:

```
DIM A[15]
DIM B$[2,6,8], C[3,1,7,3], D[8]
```

UnDim Array[] { Array[] } ...

Un-dimensions an array. The array is destroyed, releasing all of the memory it used. Multiple arrays can be destroyed with one **UnDim** statement. Each `Array[]` is specified without any index. This command is exactly the same as **Array.delete**.

Array.average <Average_nvar>, Array[{<start>,<length>}]

Finds the average of the values in a numeric array (`Array[]`) or array segment (`Array[start,length]`), and places the result into `<Average_nvar>`.

Array.copy SourceArray[{<start>,<length>}], DestinationArray[{ {- } <start_or_extras> }]

Copies elements of an existing `SourceArray[]` to the `DestinationArray[]`. If the Destination Array exists, some or all of the existing array is overwritten. If the Destination Array does not exist, a new array is created. The arrays may be either numeric or string arrays but they must both be of the same type.

You may copy an entire array (`SourceArray[]`) or an array segment (`SourceArray[<start>,<length>]`). Copying stops without error if it reaches the end of either the `SourceArray` or the `DestinationArray`.

If `<start>` is `< 1` it is set to 1, the first element of the `SourceArray`. If `<length>` is `< 0` it is set to 0.

If the Destination Array already exists, the optional `<start_or_extras>` parameter specifies where to start copying into the Destination Array.

If the Destination Array does not exist, the optional `<start_or_extras>` parameter specifies how many extra elements to add to the copy. If the parameter is a negative number, these elements are added to the start of the array, otherwise they are added to end of the array.

The extra elements for a new numeric array are initialized to zero. The extra elements for a new string array are initialized to the empty string, `""`.

See the Sample Program file, `f26_array_copy.bas`, for working examples of this command.

Array.delete Array[] { Array[] } ...

Does the same thing as **UnDim** `Array[]`.

Array.dims Source[] { { Dims[] } { NumDims } }

Provides information about the dimensions of the `Source[]` array parameter. The `Source[]` parameter may be a numeric or string array name with nothing in the brackets (`[]`). The array must already exist. The `Source[]` parameter is required, and both of the other parameters are optional.

The dimensions of the Source[] array are written to the Dims[] array, if you provide one. The Dims[] parameter must be a numeric array name with nothing in the brackets ("[]"). If the Dims[] array exists, it is overwritten. Otherwise a new array is created. The result is always a one-dimensional array.

The number of dimensions of the Source[] array is written to the NumDims parameter, if you provide one. NumDims must be a numeric variable. This value is the length of the Dims[] array.

Array.fill Array[{<start>,<length>}], <exp>

Fills an existing array or array segment with a value. The types of the array and value must match.

Array.length <length_nvar>, Array[{<start>,<length>}]

Places the number of elements in an entire array (Array[] or Array\$[]) or an array segment (Array[start,length] or Array\$[start,length]) into <length_nvar>.

Array.load Array[], <exp>, ...

Creates a new array, evaluates the list of expressions "<exp>, ...", and loads values into the new array. Specify the array name with no index(es). The array has one dimension; its size is the same as the number of expressions in the list. If the named array already exists, it is overwritten.

The array may be numeric (Array[]) or string (Array\$[]), and the expressions must be the same type as the array.

The list of expressions may be continued onto the next line by ending the line with the "~" character. The "~" character may be used between <exp> parameters, where a comma would normally appear. The "~" itself separates the parameters; the comma is optional.

The "~" character may not be used to split a parameter across multiple lines.

Examples:

```
Array.load Numbers[], 2, 4, 8 , n^2, 32
Array.load Hours[], 3, 4,7,0, 99, 3, 66~    % comma not required before ~
37, 66, 43, 83,~                          % comma is allowed before ~
83, n*5, q/2 +j
Array.load Letters$[], "a", "b","c",d$,"e"
```

Array.max <Max_nvar> Array[{<start>,<length>}]

Finds the maximum value in a numeric array (Array[]) or array segment (Array[start,length]), and places the result into the numeric variable <max_nvar>.

Array.min <Min_nvar>, Array[{<start>,<length>}]

Finds the minimum value in a numeric array (Array[]) or array segment (Array[start,length]), and places the result into the numeric variable <min_nvar>.

Array.reverse Array[{<start>,<length>}]

Reverses the order of values in a numeric or string array (Array[] or Array\$[]) or array segment (Array[start,length] or Array\$[start,length]).

Array.search Array[{<start>,<length>}], <value_exp>, <result_nvar>{<start_nexp>}

Searches in the numeric or string array (Array[] or Array\$[]) or array segment (Array[start,length] or Array\$[start,length]) for the specified numeric or string value, which may be an expression. If the value is found in the array, its position will be returned in the result numeric variable <result_nvar>. If the value is not found the result will be zero.

If the optional start expression parameter is present, the search will start at the specified element. The default value is 1.

Array.shuffle Array[{<start>,<length>}]

Randomly shuffles the values of the specified array (Array[] or Array\$[]) or array segment (Array[start,length] or Array\$[start,length]).

Array.sort Array[{<start>,<length>}]

Sorts the values of the specified array (Array[] or Array\$[]) or array segment (Array[start,length] or Array\$[start,length]) in ascending order.

Array.std_dev <sd_nvar>, Array[{<start>,<length>}]

Finds the standard deviation of the values in a numeric array (Array[]) or array segment (Array[start,length]), and places the result into the numeric variable <sd_nvar>.

Array.sum <sum_nvar>, Array[{<start>,<length>}]

Finds the sum of the values in a numeric array (Array[]) or array segment (Array[start,length]), and then places the result into the numeric variable <sum_nvar>.

Array.variance <v_nvar>, Array[{<start>,<length>}]

Finds the variance of the values in a numeric array (Array[]) or array segment (Array[start,length]), and places the result into the numeric variable <v_nvar>.

Data Structures and Pointers in BASIC!

BASIC! offers commands that facilitate working with Data Structures in ways that are not possible with traditional Basic implementations. These commands provide for the implementation of Lists, Bundles, Stacks and Queues.

What is a Pointer

The central concept behind the implementation of these commands (and many other BASIC! commands) is the *pointer*. A pointer is a numeric value that is an index into a list or table of things.

Do not confuse the pointer with the thing it points to. A pointer to a List is not a List; a pointer to a bitmap is not a bitmap. A pointer is just a number that represents something else.

As an example of pointers think of a file cabinet drawer with folders in it. That file cabinet is maintained by your administrative assistant. You never see the file drawer itself. In the course of your work you will create a new folder into which you put some information. You then give the folder to your assistant to be placed into the drawer. The assistant puts a unique number on the folder and gives you a slip of paper with that unique number on it. You can later retrieve that folder by asking your assistant to bring you the folder with that particular number on it.

In BASIC! you create an information object (folder). You then give that information object to BASIC! to put into a virtual drawer. BASIC! will give you a unique number—a pointer—for that information object. You then use that pointer to retrieve that particular information object.

Continuing with the folder analogy, let's assume that you have folders that contain information about customers. This information could be things such as name, address and phone number. The number that your assistant will give you when filing the folder will become the customer's customer number. You can retrieve this information about any customer by asking the assistant to bring you the folder with the unique customer number. In BASIC! you would use a Bundle to create that customer information object (folder). The pointer that BASIC! returns when you create the customer Bundle becomes the customer number.

Now let's assume that a customer orders something. You will want to create a Bundle that contains all the order information. Such bundles are used by the order fulfillment department, the billing department and perhaps even the marketing department (to SPAM the customer about similar products). Each Bundle could contain the item ordered, the price, etc. The Bundle will also need to contain information about the customer. Rather than replicate the customer information you will just create a customer number field that contains the customer number (pointer). The pointer that gets returned when you create the order bundle becomes the Order Number. You can create different lists of bundles for use by different departments.

It would also be nice to have a list of all orders made by a customer in the customer Bundle. You would do this by creating a List of all order numbers for that customer. When you create the customer bundle, you would ask BASIC! to create an empty List. BASIC! will return a pointer to this empty List. You would then place this pointer into the customer record. Later when the customer places an order, you will retrieve that list pointer and add the order number to the List.

You may also want to create several other Lists of order Bundles for other purposes. You may, for example, have one List of orders to be filled, another List of filled orders, another List of returned orders, another List for billing, etc. All of these Lists would simply be lists of order numbers. Each order number would point to the order Bundle which would point to the Customer Bundle.

If you were to actually create such a database in BASIC!, you would probably want to save all these Bundles and Lists onto external storage. Getting that information from the internal data structures to external storage is an exercise left to the user for now.

There are things besides List, Bundle, and Stack data structures that are accessed through pointers. These include bitmaps and graphical objects, described below in the **Graphics** section, audio clips, described in **SoundPool**, and other things.

Lists

A List is similar to a single-dimension array. The difference is in the way a List is built and used. An array must be dimensioned before being used. The number of elements to be placed in the array must be predetermined. A List starts out empty and grows as needed. Elements can be removed, replaced and inserted anywhere within the list.

There is no fixed limit on the size or number of lists. You are limited only by the memory of your device.

Another important difference is that a List is not a variable type. A numeric pointer is returned when a list is created. All further access to the List is by means of that numeric pointer. One implication of this is that it is easy to make a List of Lists. A List of Lists is nothing more than a numeric list containing numeric pointers to other lists.

Lists may be copied into new Arrays. Arrays may be added to Lists.

All of the List commands are demonstrated in the Sample Program file, f27_list.bas.

List Commands

List.create N/S, <pointer_nvar>

Creates a new, empty list of the type specified by the N or S parameter. A list of strings will be created if the parameter is **S**. A list of numbers will be created if the parameter is **N**. Do not put quotation marks around the N or S.

The pointer to the new list will be returned in the <pointer_nvar> variable.

The newly created list is empty. The size returned for a newly created list is zero.

List.add <pointer_nexp>{ <exp> }...

Adds the values of the expressions <exp>... to specified list. The expressions must all be the same type (numeric or string) as the list.

The list of <exp>s may be continued onto the next line by ending the line with the "~" character. The "~" character may be used between <exp> parameters, where a comma would normally appear. The "~" itself separates the parameters; the comma is optional.

The "~" character may not be used to split a parameter across multiple lines.

Examples:

```
List.add Nlist, 2, 4, 8 , n^2, 32
```

```
List.add Hours, 3, 4,7,0, 99, 3, 66~      % comma not required before ~  
37, 66, 43, 83,~                          % comma is allowed before ~  
83, n*5, q/2 +j
```

```
List.add Name~  
"Bill", "Jones"~  
"James", "Barnes"~  
"Jill", "Hanson"
```

List.add.list <destination_list_pointer_nexp>, <source_list_pointer_nexp>

Appends the elements in the source list to the end of the destination list.

The two lists must be of the same type (string or numeric).

List.add.array <list_pointer_nexp>, Array[{<start>,<length>}]

Appends the elements of the specified array (Array[]) or array segment (Array[start,length]) to the end of the specified list.

The Array type must be the same as the list type (string or numeric).

List.replace <pointer_nexp>, <index_nexp>, <sexp>|<nexp>

The List element specified by <index_nexp> in the list pointed to by <pointer_nexp> is replaced by the value of the string or numeric expression.

The index is one-based. The first element of the list is 1.

The replacement expression type (string or numeric) must match the list type.

List.insert <pointer_nexp>, <index_nexp>, <sexp>|<nexp>

Inserts the <sexp> or <nexp> value into the list pointed to by <pointer_nexp> at the index point <index_nexp>. If the index point is one more than the current size of the list, the new item is added at the end of the list.

The index is ones based. The first element of the list is 1.

The inserted element type must match the list type (string or numeric).

List.remove <pointer_nexp>,<index_nexp>

Removes the list element specified by <index_nexp> from the list pointed to by <pointer_nexp>.

The index is ones based. The first element of the list is 1.

List.get <pointer_nexp>, <index_nexp>, <var>

The list element specified by <index_nexp> in the list pointed to by <pointer_nexp> is returned in the specified string or numeric variable <var>.

The index is one-based. The first element of the list is 1.

The return element variable type must match the list type (string or numeric).

List.type <pointer_nexp>, <svar>

The type of list pointed to by the list pointer is returned in the string variable <svar>.

- Returns the upper case character "S" if the list is a list of strings.
- Returns the upper case character "N" if the list is a list of numbers.

List.size <pointer_nexp>, <nvar>

The size of the list pointed to by the list pointer is returned in the numeric variable <nvar>.

List.clear <pointer_nexp>

Clears the list pointed to by the list pointer and sets the list's size to zero.

List.search <pointer_nexp>, value/value\$, <result_nvar>{<start_nexp>}

Searches the specified list for the specified string or numeric value. The position of the first (left-most) occurrence is returned in the numeric variable <result_nvar>. If the value is not found in the list then the result is zero.

If the optional start expression parameter is present, the search starts at the specified element. The default start position is 1.

List.toArray <pointer_nexp>, Array\$[] / Array[]

Copies the list pointed to by the list pointer into an array. The array type (string or numeric) must be the same as the list type. If the array exists, it is overwritten, otherwise a new array is created. The result is always a one-dimensional array.

Bundles

A Bundle is a group of values collected together into a single object. A bundle object may contain any number of string and numeric values. There is no fixed limit on the size or number of bundles. You are limited only by the memory of your device.

The values are set and accessed by keys. A key is string that identifies the value. For example, a bundle might contain a person's first name and last name. The keys for accessing those name strings could be "first_name" and "last_name". An age numeric value could also be placed in the Bundle using an "age" key.

A new, empty bundle is created by using the **Bundle.create** command. The command returns a pointer to the empty bundle. Because the bundle is represented by a pointer, bundles can be placed in lists and

arrays. Bundles can also be contained in other bundles. This means that the combination of lists and bundles can be used to create arbitrarily complex data structures.

After a bundle is created, keys and values can be added to the bundle using the **Bundle.put** command. Those values can be retrieved using the keys in the **Bundle.get** command. There are other bundle commands to facilitate the use of bundles.

Bundle Auto-Create

Every bundle command except **Bundle.create** has a parameter, the `<pointer_nexp>`, which can point to a bundle. If the expression value points to a bundle, the existing bundle is used. If it does not, and the expression consists only of a single numeric variable, then a new, empty bundle is created, and the variable value is set to point to the new bundle.

That may seem complex, but it isn't, really. If there is a bundle, use it. If there is not, try to create a new one – but BASIC! can't create a new bundle if you don't give it a variable name. BASIC! uses the variable to tell you how to find the new bundle.

<code>BUNDLE.PUT b, "key1", 1.2</code>	% try to put a value in the bundle pointed to by b
<code>BUNDLE.PUT 10, key2\$, value2</code>	% try to put a value in the 10th bundle created
<code>BUNDLE.REMOVE c + d, key\$[3],</code>	% try to remove a key/value pair from a bundle
	% pointed to by c + d

In the first example, if the value of **b** points to a bundle, the **Bundle.put** puts "key1" and the value 1.2 into that bundle. If **b** is a new variable, its value is 0.0, so it does not point to a bundle. In that case, the **Bundle.put** creates a new bundle, puts "key1" and the value 1.2 into the new bundle, and sets **b** to point to the new bundle.

In the second example, if there are at least ten bundles, then the **Bundle.put** tries to put the key named in the variable **key2\$** and the value of the variable **value2** into bundle 10. If there is no bundle 10, then the command does nothing. It can't create a new variable because you did not provide a variable to return the bundle pointer.

In the third example, the bundle pointer is the value of the expression **c + d**. If there is no such bundle, the command does nothing. To create a new bundle, the bundle pointer expression must be a single numeric variable.

Bundle Commands

Bundle.create <pointer_nvar>

A new, empty bundle is created. The bundle pointer is returned in `<pointer_nvar>`.

Example:

```
BUNDLE.CREATE bptr
```


Bundle.put <pointer_nexp>, <key_sexp>, <value_nexp>|<value_sexp>

The value expression will be placed into the specified bundle using the specified key. If the bundle does not exist, a new one may be created.

The type of the value will be determined by the type of the value expression.

Example:

```
BUNDLE.PUT bptr, "first_name", "Frank"
BUNDLE.PUT bptr, "age", 44
```

Bundle.get <pointer_nexp>, <key_sexp>, <nvar>|<svar>

Places the value specified by the key string expression into the specified numeric or string variable. The type (string or numeric) of the destination variable must match the type stored with the key. If the bundle does not exist or does not contain the requested key, the command generates a run-time error.

Example:

```
BUNDLE.GET bptr, "first_name", first_name$
BUNDLE.GET bptr, "age", age
```

Bundle.keys <bundle_ptr_nexp>, <list_ptr_nexp>

Returns a list of the keys currently in the specified bundle.

The bundle pointer parameter <bundle_ptr_nexp> specifies the bundle from which to get the keys. If the bundle does not exist, a new one may be created.

The list pointer parameter <list_ptr_next> specifies the list into which to write the keys. The previous contents of the list are discarded. If the parameter does not specify a valid string list to reuse, and the parameter is a string variable, a new list is created and a pointer to the list is written to the variable.

The key names in the returned list may be extracted using the various list commands.

Example:

```
BUNDLE.KEYS bptr, list
LIST.SIZE list, size
FOR i = 1 TO size
    LIST.GET list, i, key$
    BUNDLE.TYPE bptr, key$, type$
    IF type$ = "S"
        BUNDLE.GET bptr, key$, value$
        PRINT key$, value$
    ELSE
        BUNDLE.GET bptr, key$, value
        PRINT key$, value
    ENDIF
NEXT i
```

Bundle.contains <pointer_nexp>, <key_sexp>, <contains_nvar>

If the key specified in the key string expression is contained in the bundle's keys then the "contains" numeric variable will be returned with a non-zero value. The value returned will be zero if the key is not in the bundle. If the bundle does not exist, a new one may be created.

Bundle.type <pointer_nexp>, <key_sexp>, <type_svar>

Returns the value type (string or numeric) of the specified key in the specified string variable. The <type_svar> will contain an uppercase "N" if the type is numeric. The <type_svar> will contain an uppercase "S" if the type is a string. If the bundle does not exist or does not contain the requested key, the command generates a run-time error.

Example:

```
BUNDLE.TYPE bptr, "age", type$  
PRINT type$    % will print N
```

Bundle.remove <pointer_nexp>, <key_sexp>

Removes the key named by the string expression <key_sexp>, along with the associated value, from the bundle pointed to by the numeric expression <pointer_nexp>. If the bundle does not contain the key, nothing happens. If the bundle does not exist, a new one may be created.

Bundle.clear <pointer_nexp>

The bundle pointed to by <pointer_nexp> will be cleared of all tags. It will become an empty bundle. If the bundle does not exist, a new one may be created.

Stacks

Stacks are like a magazine for a gun.



The last bullet into the magazine is the first bullet out of the magazine. This is also what is true about stacks. The last object placed into the stack is the first object out of the stack. This is called LIFO (Last In First Out).

An example of the use of a stack is the BASIC! Gosub command. When a Gosub command is executed the line number to return to is "pushed" onto a stack. When a return is executed the return line number is "popped" off of the stack. This methodology allows Gosubs to be nested to any level. Any return statement will always return to the line after the last Gosub executed.

A running example of Stacks can be found in the Sample Program file, f29_stack.bas.

There is no fixed limit on the size or number of stacks. You are limited only by the memory of your device.

Stack Commands

Stack.create N/S, <ptr_nvar>

Creates a new stack of the designated type (N=Number, S=String). The stack pointer is in <ptr_nvar>.

Stack.push <ptr_nexp>, <nexp>/<sexp>

Pushes the <nexp> or <sexp> onto the top of the stack designated by <ptr_nexp>.

The type of value expression pushed must match the type of the created stack.

Stack.pop <ptr_nexp>, <nvar>/<svar>

Pops the top-of-the-stack value designated by <ptr_nexp> and places it into the <nvar> or <svar>.

The type of the value variable must match the type of the created stack.

Stack.peek <ptr_nexp>, <nvar>/<svar>

Returns the top-of-stack value of the stack designated by <ptr_nexp> into the <nvar> or <svar>. The value will remain on the top of the stack.

The type of the value variable must match the type of the created stack.

Stack.type <ptr_nexp>, <svar>

The type (numeric or string) of the stack designated by <ptr_nexp> will be returned in <svar>. If the stack is numeric, the upper case character "N" will be returned. If the stack is a string stack, the upper case character "S" will be returned.

Stack.isEmpty <ptr_nexp>, <nvar>

If the stack designated by <ptr_nexp> is empty the value returned in <nvar> will be 1. If the stack is not empty the value will be 0.

Stack.clear <ptr_nexp>

The stack designated by <ptr_nexp> will be cleared.

Queues

A Queue is like the line that forms at your bank. When you arrive, you get in the back of the line or queue. When a teller becomes available the person at the head of the line or queue is removed from the queue to be serviced by the teller. The whole line moves forward by one person. Eventually, you get to the head of the line and will be serviced by the next available teller. A queue is something like a stack except the processing order is First In First Out (FIFO) rather than LIFO.

Using our customer order processing analogy, you could create a queue of order bundles for the order processing department. New order bundles would be placed at the end of the queue. The top-of-the-

queue bundle would be removed by the order processing department when it was ready to service a new order.

There are no special commands in BASIC! for Queue operations. If you want to make a queue, create a list.

Use **List.add** to add new elements to the end of the queue.

Use **List.get** to get the element at the top of the queue and use **List.remove** to remove that top of queue element. You should, of course, use **List.size** before using **List.get** to ensure that there is a queued element remaining

Comments

! - Single Line Comment

If the first character in a line is the "!" character, BASIC! considers the entire line a comment and ignores it. If the "!" appears elsewhere in the line it does not indicate a comment.

Rem - Single Line Comment (legacy)

If the first three characters in a line are "Rem", "REM", or even "rEm", BASIC! considers the entire line a comment and ignores it. If "Rem" appears elsewhere in the line it does not indicate a comment.

!! - Block Comment

When a line begins with the "!!" characters, all lines that follow are considered comments and are ignored by BASIC! The Block Comment section ends at the next line that starts with "!!"

% - Middle of Line Comment

If the "%" character appears in a line (except within a quoted string) then rest of the line is a comment.

Expressions

Numeric Expression <nexp>

A numeric expression consists of one or more numeric variables or numeric constants separated by binary operators and optionally preceded by unary operators. The definition can be stated more completely using this standard formal notation:

<nexp> := {<numeric variable> | <numeric constant> } {<noperator> <nexp> }

The next few sections define all of the terms.

Numeric Operators <operator>

The numeric operators are listed by precedence. Higher precedence operators are executed before lower precedence operators. Precedence can be changed by using parentheses.

1. Unary +, Unary –
2. Exponent ^
3. Multiply *, Divide /
4. Add +, Subtract –

Note that the comma (',') is not an operator in BASIC!. It is sometimes uses as a separator between expressions; for example, see the **PRINT** command.

Numeric Expression Examples

```
a
a*b + 4/d - 2*(d^2)
a + b + d + RND()
b + CEIL(d/25) + 5
```

Pre- and Post-Increment Operators

++x	Increments the value of x by 1 before the x value is used
--y	Decrements the value of y by 1 before the y value is used
x++	Increments the value of x by 1 after the x value is used
y--	Decrements the value of y by 1 after the y value is used

```
a = 5          % creates the variable a and sets it to 5
PRINT --a      % sets a to 4 and prints 4
PRINT a--      % prints 4 and sets a to 3
```

These operations work only on numeric variables. Their action is performed as part of evaluating the variable, so they do not follow normal precedence rules.

Using these operators on a variable makes the variable unavailable for other operations that require a variable. For example, you cannot pass a variable by reference (see **User-Defined functions**) if you pre- or post-increment or -decrement it, because you cannot pass an expression by reference. An exception is made to allow implicit assignment (actual or implied **LET**).

String Expression <sexp>

A string expression consists of one or more string variables or string constants separated by '+' operators. The definition can be stated more completely using this standard formal notation:

<sexp> := {<string variable>|<string constant>} { + <sexp> }

There is only one string operator: +. This is the concatenation operator. It is used to join two strings:

```
PRINT "abc" + "def"  % prints abcdef
```

Logical Expression <lexp>

Logical expressions, or Boolean expressions, produce only two results: false or true. False is represented in BASIC! by the numeric value of zero. Anything that is not zero is true. False = 0. True = not 0.

There are two types of logical expressions: Numeric logical expressions and string logical expressions. Both types produce a numerically-represented values of true or false. Each type consists of one or more variables or constants separated by binary logical operators, formally defined like this:

<slexp> := {<string variable>|<string constant>} <logical operator> {<string variable>|<string constant>}

<nlexp> := {<numeric variable>|<numeric constant>} <logical operator> {<numeric variable>|<numeric constant>}

There is also the unique unary NOT (!) operator. NOT inverts the truth of a logical expression.

Logical Operators

Most of the logical operators are used for comparison. You can compare strings or numbers (<, =, etc.). You can use the other Boolean operators (!, &, |) on numbers but not on strings.

This table shows all of the logical operators. They are listed by precedence with the highest precedence first. All of these operators have *lower* precedence than any of the numeric operators or the one string operator. Precedence may be modified by using parentheses.

Precedence	Operator	Meaning	Operands
1	<	Less Than	Two <nlexp> or two <slexp>
	>	Greater Than	
	<=	Less Than or Equal	
	>=	Greater Than or Equal	
	=	Equal	
	<>	Not Equal	
2	!	Unary Not	One <nlexp> only
3	&	And	Two <nlexp> only
4		Or	Two <nlexp> only

Examples of Logical Expressions

```
1 < 2 (true)
3 <> 4 (true)
"a" < "bcd" (true)
1 & 0 (false)
!(1 & 0) (true)
```

Parentheses

Parentheses can be used to override operator precedence.

```
a = b * c + d      % the multiplication is done first
a = b * (c + d)    % the addition is done first
```

Parentheses can also be placed around a variable, anywhere except to the left of an = sign. This can be useful in places where BASIC! may mistake part of a variable for a special keyword. For an example, see **Program Control Commands – For - To - Step / Next**, below.

Assignment Operations

Variables get values by means of assignment statements. Simple assignment statements are of the form:

```
<nvar> = <nexp>
<svar> = <ssexp>
```

The special form of the statement allows BASIC! to infer the command. The implied command is **LET**.

Let

The original Basic language used the command, **LET**, to denote an assignment operation as in:

```
LET <nvar> = <nexp>
```

BASIC! also has the **LET** command but it is optional. If you use other programming languages, it may look strange to you, but there are two reasons you might use **LET**.

First, you must use **LET** if you want to have a variable name start with a BASIC! keyword. Such keywords may not appear at the beginning of a new line. The statement:

```
Letter$ = "B"
```

is seen by BASIC! as

```
LET ter$ = "B"
```

If you really want to use Letter\$ as a variable, you can safely use it by putting it in a **LET** statement:

```
LET Letter$ = "B"
```

If you do the assignment in a single-line **IF** statement, you must also use the **LET** command:

```
IF 1 < 2 THEN LET letter$ = "B"
```

Second, assignment is faster with the **LET** command than without it.

OpEqual Assignment Operations

All of the binary arithmetic and logical operators (+, -, *, /, ^, &, |) may be used with the equals sign (=) to make a single "OpEqual" operator. The combined operator works like this:

```
var op= expression is the same as var = var op (expression)
```

Here are some examples:

a += 1	is the same as	a = a + 1
a\$ += "xyz"	is the same as	a\$ = a\$ + "xyz"
b /= 5 + 3	is the same as	b = b / (5 + 3)
c ^= log(37) + 1	is the same as	c = c ^ (log(37) + 1)
d *= --d + d--	is the same as	d = d * (--d + d--)
m &= (x\$ = y\$) (x\$!= z\$) is the same as m = m & ((x\$ = y\$) (x\$!= z\$))		

Math Functions

Math functions act like numeric variables in a <nexp> (or <lexp>).

BOR(<nexp1>, <nexp2>)

Returns the logical bitwise value of <nexp1> OR <nexp2>. The double-precision floating-point values are converted to 64-bit integers before the operation.

BOR(1,2) is 3

BAND(<nexp1>, <nexp2>)

Returns the logical bitwise value of <nexp1> AND <nexp2>. The double-precision floating-point values are converted to 64-bit integers before the operation.

BAND(3,1) is 1

BXOR(<nexp1>, <nexp2>)

Returns the logical bitwise value of <nexp1> XOR <nexp2>. The double-precision floating-point values are converted to 64-bit integers before the operation.

BXOR(7,1) is 6

BNOT(<nexp>)

Returns the bitwise complement value of <nexp>. The double-precision floating-point value is converted to a 64-bit integer before the operation.

BNOT(7) is -8
 HEX\$(BNOT(HEX("1234"))) is ffffffffedcb

ABS(<nexp>)

Returns the absolute value of <nexp>.

SGN(<nexp>)

Returns the signum function of the numerical value of <nexp>, representing its sign.

When the value is:	Return:
> 0	1

= 0	0
< 0	-1

RANDOMIZE({<nexp>})

Creates a pseudo-random number generator for use with the **RND()** function. The optional seed parameter <nexp> initializes the generator. Omitting the parameter is the same as specifying 0. If you call **RND()** without first calling **RANDOMIZE()**, it is the same as if you had executed **RANDOMIZE(0)**.

A non-zero seed initializes a predictable series of pseudo-random numbers. That is, for a given non-zero seed value, subsequent **RND()** calls will always return the same series of values.

If the seed is 0, the sequence of numbers from **RND()** is unpredictable and not reproducible. However, repeated **RANDOMIZE(0)** calls do not produce "more random" sequences.

The **RANDOMIZE()** function always returns zero.

RND()

Returns a random number generated by the pseudorandom number generator. If a **RANDOMIZE(n)** has not been previously executed then a new random generator will be created using **RANDOMIZE(0)**.

The random number will be greater than or equal to zero and less than one. ($0 \leq n < 1$).

```
d = FLOOR(6 * RND() + 1)           % roll a six-sided die
```

MAX(<nexp>, <nexp>)

Returns the maximum of two numbers as an <nvar>.

MIN(<nexp>, <nexp>)

Returns the minimum of two numbers as an <nvar>.

CEIL(<nexp>)

Rounds up towards positive infinity. 3.X becomes 4 and -3.X becomes -3.

FLOOR(<nexp>)

Rounds down towards negative infinity. 3.X becomes 3 and -3.X becomes -4.

INT(<nexp>)

Returns the integer part of <nexp>. 3.X becomes 3 and -3.X becomes -3. This operation may also be called truncation, rounding down, or rounding toward zero.

FRAC(<nexp>)

Returns the fractional part of <nexp>. 3.4 becomes 0.4 and -3.4 becomes -0.4.

FRAC(n) is equivalent to "n - INT(n)".

MOD(<nexp1>, <nexp2>)

Returns the remainder of <nexp1> divided by <nexp2>. If <nexp2> is 0, the function generates a runtime error.

ROUND(<value_nexp>{, <count_nexp>{, <mode_sexp>}})

In its simplest form, **ROUND(<value_nexp>)**, this function returns the closest whole number to <nexp>. You can use the optional parameters to specify more complex operations.

The <count_nexp> is an optional decimal place count. It sets the number of places to the right of the decimal point. The last digit is rounded. The decimal place count must be ≥ 0 . Omitting the parameter is the same as setting it to zero.

The <mode_sexp> is an optional rounding mode. It is a one- or two-character mnemonic code that tells **ROUND()** what kind of rounding to do. It is not case-sensitive. There are seven rounding modes:

Mode:	Meaning:	-3.8	-3.5	-3.1	3.1	3.5	3.8
"HD"	Half-down	-4.0	-3.0	-3.0	3.0	3.0	4.0
"HE"	Half-even	-4.0	-4.0	-3.0	3.0	4.0	4.0
"HU"	Half-up	-4.0	-4.0	-3.0	3.0	4.0	4.0
"D"	Down	-3.0	-3.0	-3.0	3.0	3.0	3.0
"U"	Up	-4.0	-4.0	-4.0	4.0	4.0	4.0
"F"	Floor	-4.0	-4.0	-4.0	3.0	3.0	3.0
"C"	Ceiling	-3.0	-3.0	-3.0	4.0	4.0	4.0

In this table, "down" means "toward zero" and "up" means "away from zero" (toward $\pm\infty$)

"Half" refers to behavior when a value is half-way between rounding up and rounding down (x.5 or -x.5). "Half-down" rounds x.5 towards zero and "half-up" rounds x.5 away from zero.

"Half-even" is either "half-down" or "half-up", whichever would make the result **even**. 4.5 and 3.5 both round to 4.0. "Half-even" is also called "banker's rounding", because it tends to average out rounding errors.

If you do not provide a <mode_sexp>, **ROUND()** adds +0.5 and rounds down (toward zero). This is legacy behavior, copied from earlier versions of BASIC!. **ROUND(n)** is NOT the same as **ROUND(n, 0)**.

ROUND() generates a runtime error if <count_nexp> < 0 or <mode_sexp> is not valid.

Examples:

pi = ROUND(3.14159)	% pi is 3.0
pi = ROUND(3.14159, 2)	% pi is 3.14
pi = ROUND(3.14159, , "U")	% pi is 4.0
pi = ROUND(3.14159, 4, "F")	% pi is 3.1415
negpi = ROUND(-3.14159, 4, "D")	% negpi is -3.1416

Note that **FLOOR(n)** is exactly the same as **ROUND(n, 0, "F")**, but **FLOOR(n)** is a little faster. In the same way, **CEIL(n)** is the same as **ROUND(n, 0, "C")**, and **INT(n)** is the same as **ROUND(n, 0, "D")**.

SQR(<nexp>)

Returns the closest double-precision floating-point approximation of the positive square root of <nexp>. If the value of <nexp> is negative, the function generates a runtime error.

CBRT(<nexp>)

Returns the closest double-precision floating-point approximation of the cube root of <nexp>.

LOG(<nexp>)

Returns the natural logarithm (base e) of <nexp>.

LOG10(<nexp>)

Returns the base 10 logarithm of the <nexp>.

EXP(<nexp>)

Returns e raised to the <nexp> power.

POW(<nexp1>, <nexp2>)

Returns <nexp1> raised to the <nexp2> power.

HYPOT(<nexp_x>, <nexp_y>)

Returns $\text{SQR}(x^2+y^2)$ without intermediate overflow or underflow.

PI()

Returns the double-precision floating-point value closest to pi.

SIN(<nexp>)

Returns the trigonometric sine of angle <nexp>. The units of the angle are radians.

COS(<nexp>)

Returns the trigonometric cosine of angle <nexp>. The units of the angle are radians.

TAN(<nexp>)

Returns the trigonometric tangent of angle <nexp>. The units of the angle are radians.

SINH(<nexp>)

Returns the trigonometric hyperbolic sine of angle <nexp>. The units of the angle are radians.

COSH(<nexp>)

Returns the trigonometric hyperbolic cosine of angle <nexp>. The units of the angle are radians.

ASIN(<nexp>)

Returns the arc sine of the angle <nexp>, in the range of $-\pi/2$ through $\pi/2$. The units of the angle are radians. If the value of <nexp> is less than -1 or greater than 1, the function generates a runtime error.

ACOS(<nexp>)

Returns the arc cosine of the angle <nexp>, in the range of 0.0 through π . The units of the angle are radians. If the value of <nexp> is less than -1 or greater than 1, the function generates a runtime error.

ATAN(<nexp>)

Returns the arc tangent of the angle <nexp>, in the range of $-\pi/2$ through $\pi/2$. The units of the angle are radians.

ATAN2(<nexp_y>, <nexp_x>)

Returns the angle *theta* from the conversion of rectangular coordinates (*x*, *y*) to polar coordinates (*r*, *theta*). (Please note the order of the parameters in this function.)

TODEGREES(<nexp>)

Converts <nexp> angle measured in radians to an approximately equivalent angle measured in degrees.

TORADIANS(<nexp>)

Converts <nexp> angle measured in degrees to an approximately equivalent angle measured in radians.

VAL(<sexp>)

Returns the numerical value of the string expression <sexp> interpreted as a signed decimal number. If the string is empty (""), or does not represent a number, the function generates a runtime error.

- A sign ("+" or "-"), a decimal point ("." only), and an exponent (power of 10) are optional.
- An exponent is "e" or "E" followed by a number. The number may have a sign but no decimal point.
- The string may have leading and/or trailing spaces, but no spaces between any other characters.

IS_NUMBER(<sexp>)

Tests a string expression <sexp> in the same way as **VAL()** and returns a logical value:

- TRUE (non-zero) if **VAL()** would successfully convert the string to a number
- FALSE (0) if **VAL()** would generate a run-time error.

For example, **VAL("name")** generates a run-time error, but **IS_NUMBER("name")** returns FALSE.

If **VAL()** would report a syntax error, **IS_NUMBER()** reports a syntax error.

For example, **IS_NUMBER()**, **IS_NUMBER(num)**, and **IS_NUMBER(5)** are syntax errors.

LEN(<sexp>)

Returns the length of the <sexp>.

HEX(<sexp>)

Returns the numerical value of the string expression <sexp> interpreted as a hexadecimal integer. The characters of the string can be only hexadecimal digits (0-9, a-h, or A-H), with an optional leading sign ("+" or "-"), or the function generates a runtime error.

OCT(<sexp>)

Returns the numerical value of the string expression <sexp> interpreted as an octal integer. The characters of the string can be only octal digits (0-7), with an optional leading sign ("+" or "-"), or the function generates a runtime error.

BIN(<sexp>)

Returns the numerical value of the string expression <sexp> interpreted as a binary integer. The characters of the string can be only binary digits (0 or 1), with an optional leading sign ("+" or "-"), or the function generates a runtime error.

SHIFT(<value_nexp>, <bits_nexp>)

Shifts the value <value_nexp> by the bit count <bits_nexp>. If the bit count is < 0, the value will be shifted left. If the bit count is > 0, the bits will be shifted right. The right shift will replicate the sign bit. The double-precision floating-point value are truncated to 64-bit integers before the operation.

ASCII(<sexp>{, <index_nexp>})

Returns the ASCII value of one character of <sexp>. By default, it is the value of the first character. You can use the optional <index_nexp> to select any character. The index of the first character is 1.

A valid ASCII value is between 0 and 255. If <sexp> is an empty string (""), the value returned will be 256 (one more than the largest 8-bit ASCII value). For non-ASCII Unicode characters, **ASCII()** returns invalid values; use **UCODE()** instead.

UCODE(<sexp>{, <index_nexp>})

Returns the Unicode value of one character of <sexp>. By default, it is the value of the first character. You can use the optional <index_nexp> to select any character. The index of the first character is 1.

If <sexp> is an empty string (""), the value returned will be 65536 (one more than the largest 16-bit Unicode value). If the selected character of <sexp> is a valid ASCII character, this function returns the same value as **ASCII()**.

IS_IN(<sub_sexp>, <base_sexp>{, <start_nexp>})

Returns the position of an occurrence of the substring <sub_sexp> in the base string <base_sexp>.

If the optional start parameter <start_nexp> is not present then the function starts at the first character and searches forward.

If the start parameter is ≥ 0 , then it is the starting position of a forward (left-to-right) search. The left-most character is position 1. If the parameter is negative, it is the starting position of a reverse (right-to-left) search. The right-most character is position -1.

If the substring is not in the base string, the function returns 0. It can not return a value larger than the length of the base string.

STARTS_WITH(<sub_sexp>, <base_sexp>{, <start_nexp>})

Determines if the substring <sub_sexp> exactly matches the part of the base string <base_sexp> that starts at the position <start_nexp>. The <start_nexp> parameter is optional; if it is not present then the default starting position is 1, the first character, so the base string must start with the substring. If present, <start_nexp> must be ≥ 1 .

The function returns the length of the matching substring. If no match is found, the function returns 0.

ENDS_WITH(<sub_sexp>, <base_sexp>)

Determines if the substring <sub_sexp> exactly matches the end of the base string <base_sexp>.

If the base string ends with the substring, the function returns the index into the base string where the substring starts. The value will always be ≥ 1 . If no match is found, the function returns 0.

GR_COLLISION(<object_1_nvar>, <object_2_nvar>)

The variables <object_1_nvar> and <object_2_nvar> are the object pointers returned when the objects were created.

If the boundary boxes of the two objects overlap then the function will return true (not zero). If they do not overlap then the function will return false (zero).

Objects that may be tested for collision are: point, rectangle, bitmap, circle, arc, oval, and text. In the case of a circle, an arc, an oval, or text, the object's rectangular boundary box is used for collision testing, not the actual drawn object.

BACKGROUND()

A running BASIC! program continues to run when the HOME key is tapped. This is called running in the Background. When not in the Background mode, BASIC! is in the Foreground mode. BASIC! exits the Background mode and enters the Foreground mode when the BASIC! icon on the home screen is tapped.

Sometimes a BASIC! programmer wants to know if the program is running in the Background. One reason for this might be to stop music playing while in the Background mode.

The **BACKGROUND()** function returns true (1) if the program is running in the background. It returns false (0) if the program is not running in the background.

If you want to be able to detect Background mode while Graphics is open, you must not call **Gr.render** while in the Background mode. Doing so will cause the program to stop running until the Foreground mode is re-entered. Use the following code line for all **Gr.render** commands:

```
IF !BACKGROUND() THEN GR.RENDER
```

Time Functions

CLOCK()

Returns the time in milliseconds since the last boot.

TIME()

Returns the time in milliseconds since 12:00:00 AM, January 1, 1970, UTC (the "epoch"). The time interval is the same everywhere in the world, so the value is not affected by the **TimeZone** command.

TIME(<year_exp>, <month_exp>, <day_exp>, <hour_exp>, <minute_exp>, <second_exp>)

Like **TIME()**, except the parameters specify a moment in time. The specification is not complete, as it does not include the timezone. You may specify a timezone with the **TimeZone** command. If you do not specify a timezone, your local timezone is used.

The parameter expressions may be either numeric expressions or string expressions. This is an unusual aspect as it isn't allowed anywhere else in BASIC!. If a parameter is a string, then it must evaluate to a number: digits only, one optional decimal point somewhere, optional leading sign, no embedded spaces. If the string parameter does not follow the rules, BASIC! reports a syntax error, like using a string in a place that expects a numeric expression.

TIME(...) (the function) and **Time** (the command) are inverse operations. **TIME(...)** can take the first six return parameters of the **Time** command directly as input parameters.

With the **USING\$()** or **FORMAT_USING\$()** functions, you can express a moment in time as a string in many different ways, formatted for your locale.

String Functions

GETERROR\$()

Return information about a possible error condition.

An error that stops your program writes an error message to the Console. If you trap the error with the **OnError:** interrupt label, your program does not stop and the error is not printed. You can use **GETERROR\$()** to retrieve the error message.

Certain commands can report errors without stopping your program. These commands include **App.broadcast**, **App.start**, **Audio.load**, **Byte.open**, **Text.open**, **Zip.Open**, **Encrypt**, **Decrypt**, **Font.load**, **GPS.open**, **GrabFile**, **GrabURL**, **Gr.get.type**, the **ENCODE\$()** and **DECODE\$()** functions, and any command that can create a bitmap.

When you run one of these commands, you can call **GETERROR\$()** to retrieve information about the error state. For example, if **Text.open** cannot open a file, it sets the file pointer to -1, and writes a **GETERROR\$()** message such as "<filename> not found". If no error occurred, **GETERROR\$()** returns the string "No error".

Because there are commands that clear the error message, you should not expect **GETERROR\$()** to retain its message. Capture the message in a variable as soon as possible, and do not call **GETERROR\$()** again for the same error.

CHR\$(<nexp>, ...)

Return the character string represented by the values of list of numerical expressions. Each <nexp> is converted to a character. The expressions may have values greater than 255 and thus can be used to generate Unicode characters.

```
PRINT CHR$(16*4 + 3)      % Hexadecimal 43 is the character "C". This prints: C
PRINT CHR$(945, 946)      % Decimal for the characters alpha and beta: Prints: αβ
```

LEFT\$(<ssexp>, <count_nexp>)

Return the left-most characters of the string <ssexp>. The number of characters to return is set by the count parameter, <count_nexp>.

- If the count is greater than 0, return <count_nexp> characters, counting from the left.
- If the count is less than 0, return all but <count_nexp> characters. The number to return is the string length reduced by <count_nexp>: **LEFT\$(a\$, -2)** is the same as **LEFT\$(a\$, LEN(a\$) - 2)**.
- If the count is 0, return an empty string ("").
- If the count is greater than the length of the string, return the entire string.

MID\$(<ssexp>, <start_nexp>{, <count_nexp>})

Return a substring of the string <ssexp>, beginning or ending at the start position <start_nexp>. The first character of the string is at position 1. If the start position is 0 or negative, it is set to 1.

The count parameter is optional. If it is omitted, return all of the characters from the start position to the end of the string.

```
a$ = MID$("dinner", 2)      % a$ is "inner"
```


Otherwise, the absolute value of the count specifies the length of the returned substring:

- If the count is greater than 0, begin at <start_nexp> and count characters to the **right**. That is, return the substring that **begins** at the start position.
If the start position is greater than the length of the string, return an empty string ("").
- If the count is less than 0, begin at <start_nexp> and count characters to the **left**. That is, return the substring that **ends** at the start position.
If the start position is greater than the length of the string, it is set to the end of the string.
- If the count is 0, return an empty string ("").

```
a$ = MID$("dinner", 2, 3)      % a$ is "inn"
a$ = MID$("dinner", 4, -3)     % a$ is "inn"
a$ = MID$("dinner", 3, 0)      % a$ is ""
```

RIGHT\$(<sexp>, <count_nexp>)

Return the right-most characters of the string <sexp>. The number of characters to return is set by the count parameter, <count_nexp>.

- If the count is greater than 0, return <count_nexp> characters, counting from the right.
- If the count is less than 0, return all but <count_nexp> characters. The number to return is the string length reduced by <count_nexp>: **RIGHT\$(a\$, -2)** is the same as **RIGHT\$(a\$, LEN(a\$) - 2)**.
- If the count is 0, return an empty string ("").
- If the count is greater than the length of the string, return the entire string.

REPLACE\$(<sexp>, <argument_sexp>, <replace_sexp>)

Returns <sexp> with all instances of <argument_sexp> replaced with <replace_sexp>.

TRIM\$(<sexp>{, <test_sexp>})

Returns <sexp> with leading and trailing occurrences of <test_sexp> removed.

The expression to trim off, <test_sexp>, is optional. If omitted, all leading and trailing whitespace is removed. That is, the default <test_sexp> is the regular expression "\s+", which means "all whitespace". To use this regular expression in a BASIC! string, you must write it "\\s+" (escape the backslash).

As with the **WORD\$()** function and the **SPLIT** command, the <test_exp> is a regular expression. See **Split** for a note about Regular Expressions.

LTRIM\$(<sexp>{, <test_sexp>})

RTRIM\$(<sexp>{, <test_sexp>})

Exactly like **TRIM\$()**, except that **LTRIM\$()** trims only the left end of the source string <sexp>, while **RTRIM\$()** trims only the right end.

WORD\$(<source_sexp>, <n_nexp> {, <test_sexp>})

This function returns a word from a string. The <source_sexp> string is split into substrings at each location where <test_sexp> occurs. The <n_nexp> parameter specifies which substring to return; numbering starts at 1. The <test_sexp> is removed from the result. The <test_sexp> parameter is an optional Regular Expression; if it is not given, the source string is split on whitespace. Specifically, the default <test_sexp> is "\s+".

Leading and trailing occurrences of <test_sexp> are stripped from <source_sexp> before it is split. If <n_nexp> is less than 1 or greater than the number of substrings found in <source_sexp>, then an empty string ("") is returned. Two adjacent occurrences of <test_sexp> in <source_sexp> result in an empty string; <n_nexp> may select this empty string as the return value.

Examples:

```
string$ = "The quick brown fox"
result$ = WORD$(string$, 2);           % result$ is "quick"

string$ = ":a:b:c:d"
delimiter$ = ":"
SPLIT array$[,], string$, delimiter$   % array$[1] is ""
result$ = WORD$(string$, 1, delimiter$) % result$ is "a", not ""
```

This function is similar to the **Split** command. See **Split** for a note about Regular Expressions.

ENCODE\$(<type_sexp>, {<qualifier_sexp>}, <source_sexp>)

Returns the string <source_sexp> encoded in one of several ways, as specified by the <type_sexp>. The <qualifier_sexp> usage depends on the type:

Type	Qualifier	Default	Result
"ENCRYPT"	password	"" (empty)	Encrypts the source string using the password parameter. The encryption algorithm is "PBESWithMD5AndDES". This usage of ENCODE\$() works the same way as the ENCRYPT command. Use the DECODE\$() function to decrypt.
"DECRYPT"	password	""	Same as type "ENCRYPT" (encrypts, does not decrypt).
"ENCRYPT_RAW"	password	""	Same as type "ENCRYPT" except the input and output are buffer strings. This is useful for encrypting binary data.
"DECRYPT_RAW"	password	""	Same as "ENCRYPT_RAW" (encrypts, does not decrypt).
"URL"	charset	"UTF-8"	Encodes the source string using the format required by HTML application/x-www-form-urlencoded . You should omit the charset parameter.
"BASE64"	charset	"UTF-8"	Encodes the source string into the Base64 representation of binary data. See RFCs 2045 and 3548 . The simplest way to use this function is to omit the charset parameter.

The type is required, but see below for the two-parameter form of **ENCODE\$()** and **DECODE\$()**.

The type IS NOT case-sensitive: "BASE64", "base64", and "Base64" are all the same.

The qualifier is optional, but its comma is required.

If you supply the qualifier, whether password or charset, it IS case-sensitive.

If the source string cannot be encoded (or encrypted) with the specified charset (or password), the function returns an empty string (""). You can call the **GETERROR\$()** function to get an error message.

"ENCRYPT", "DECRYPT", and "URL" can be used on any BASIC! string. The string is converted to a byte stream, then the byte stream is encrypted or URL-encoded. The encrypted byte stream is converted to string format using the Base64 representation of binary data (see comment for "BASE64" in the table above. URL-encoded strings do not require this extra step.

"ENCRYPT_RAW" and "DECRYPT_RAW" are intended for use with binary data in a buffer string. A buffer string is a special use of the BASIC! string in which each 16-bit character consists of one byte of 0 and one byte of data. "ENCRYPT_RAW" can encrypt a buffer string or an ASCII text string, but using it on Unicode text will corrupt the string. After encryption, the result is returned in another buffer string.

"BASE64" also converts its string to a byte-stream before encoding it to Base64. The default conversion, using UTF-8, works with any BASIC! string; specifying another character set encoding may corrupt data.

Normally, you would use "BASE64" on binary data in a buffer string. In this case, you may specify any valid charset with no data corruption. The encoding string will change, but it can always be decoded using the same charset.

See the two-parameter form of **ENCODE\$()**, below, for a partial list of valid charsets.

DECODE\$(<type_sexp>, {<qualifier_sexp>}, <source_sexp>)

Returns the result of decoding a string <source_sexp> that was encoded with ENCODE\$(). The <type_sexp> and <qualifier_sexp> describe how the string was encoded. You must use the same type and qualifier that were used to encode the source string, or you may get unpredictable results.

Type	Qualifier	Default	Result
"ENCRYPT"	password	"" (empty)	Decrypts the source string using the password parameter. The encryption algorithm is "PBESWithMD5AndDES". This usage of DECODE\$() works the same way as the DECRYPT command.
"DECRYPT"	password	""	Same as type "ENCRYPT" (decrypts, does not encrypt).
"ENCRYPT"	password	""	Same as type "DECRYPT" except the input and output are buffer strings. This is useful for decrypting binary data.
"DECRYPT"	password	""	Same as type "ENCRYPT".
"URL"	charset	"UTF-8"	Decodes the source string assumed to be in the format required by HTML application/x-www-form-urlencoded . You should omit the charset parameter.
"BASE64"	charset	"UTF-8"	Decodes the source string holding the Base64 representation of binary data. See RFCs 2045 and 3548 .

The type is required, but see below for the two-parameter form of **ENCODE\$()** and **DECODE\$()**.

The type IS NOT case-sensitive: "BASE64", "base64", and "Base64" are all the same.

The qualifier is optional, but its comma is required.

If you supply the qualifier, whether password or charset, it IS case-sensitive.

The source string is decoded to a byte stream according to the type. Then the byte stream is converted to a BASIC! string (UTF-16) according to the character encoding (the charset parameter), which describes how to interpret the byte stream. The charset is always UTF-8 for decryption, and defaults to UTF-8 for the other types. The most common usage of this function is to omit the charset.

If the source string was encoded from binary data (with "ENCRYPT_RAW" or "BASE64"), the resulting BASIC! string will be a buffer string. When a string is used as a buffer, one byte of data is written into the lower 8 bits of each 16-bit character, and the upper 8 bits are 0. You can extract the binary data from the string, one byte at a time, using the **ASCII()** or **UCODE()** functions.

If the source string cannot be decoded (or decrypted) with the specified charset (or password), the function returns an empty string (""). You can call the **GETERROR\$()** function to get an error message.

See the two-parameter form of **ENCODE\$()**, below, for a partial list of valid charsets.

ENCODE\$(<charset_sexp>, <source_sexp>)

Encodes the string <source_sexp> using the character encoding of the <charset_sexp> and returns the result in a buffer string. When a string is used as a buffer, one byte of data is written into the lower 8 bits of each 16-bit character, and the upper 8 bits are 0.

The charset specifies the rules used to convert the source string into a byte stream. The stream is written to a buffer string, one byte per character. The bytes are not reassembled into 16-bit characters.

The charsets "UTF-8", "UTF-16", "UTF-16BE", "UTF-16LE", "US-ASCII", and "ISO-8859-1" are always available. Your device may have additional charsets. The charset names are case-sensitive, but the standard charsets have aliases for convenience. For example, "utf8" is valid.

If the source string cannot be encoded with the specified charset, the function returns an empty string (""). You can call the **GETERROR\$()** function to get an error message.

If you create a buffer string with **ENCODE\$()**, you can write the bytes to a file with **Byte.write.buffer**.

For encryption and URL- or Base64-encoding, see the three-parameter form of **ENCODE\$()**, above.

DECODE\$(<charset_sexp>, <buffer_sexp>)

Decodes the buffer string <buffer_sexp> that was encoded using the <charset_sexp> and returns the result in a standard BASIC! string. A buffer string is a special use of the BASIC! string in which each 16-bit character consists of one byte of 0 and one byte of data.

If the source string cannot be decoded with the specified charset, the function returns an empty string (""). You can call the **GETERROR\$()** function to get an error message.

If you attempt to **DECODE\$()** a string that is not a buffer string, you may get unexpected results. Besides the function **ENCODE\$()**, the commands **Byte.read.buffer** and **BT.read.bytes** can write buffer strings. Your program can also build such strings directly, character-by-character.

If you read data from a file with **Byte.read.buffer**, you can use **DECODE\$()** to reassemble the bytes into BASIC! (UTF-16) strings. The charset specifies how the original string was encoded when it was written as bytes to the file.

For example, a binary file may have embedded text strings for names or titles. In order to allow Unicode, the text may be encoded. Let's say you read 32 bytes of binary data, consisting of 8 bytes of binary and 24 bytes of UTF-8-encoded text:

```
BYTE.READ.BUFFER file, 32, bfr$  
namebfr$ = MID$(bfr$, 9)  
name$ = DECODE$("UTF-8", namebfr$)
```

For encryption and URL- or Base64-decoding, see the three-parameter form of **DECODE\$()**, above.

STR\$(<nexp>)

Returns the string representation of <nexp>.

LOWER\$(<sexp>)

Returns <sexp> in all lower case characters.

UPPER\$(<sexp>)

Returns <nexp> in all upper case characters.

VERSION\$()

Returns the version number of BASIC! as a string.

INT\$(<nexp>)

Returns a string representing the integer part of the numeric expression.

HEX\$(<nexp>)

Returns a string representing the hexadecimal representation of the numeric expression.

OCT\$(<nexp>)

Returns a string representing the octal representation of the numeric expression.

BIN\$(<nexp>)

Returns a string representing the binary representation of the numeric expression.

USING\$(<locale_sexp> , <format_sexp> { , <exp>}...)

Returns a string, using the locale and format expressions to format the expression list.

This function gives BASIC! programs access to the `Formatter` class of the Android platform. You can find full documentation here: <http://developer.android.com/reference/java/util/Formatter.html>.

The `<locale_sexp>` is a string that tells the formatter to use the formatting conventions of a specific language and region or country. For example, "en_US" specifies American English conventions.

The `<format_sexp>` is a string that contains *format specifiers*, like "%d" or "%7.2f," that tell the formatter what to do with the expressions that follow.

The format string is followed by a list of zero or more expressions. Most format specifiers take one argument from the list, in order. If you don't provide as many arguments as your format string needs, you will get a detailed Java error message.

Each expression must also match the type of the corresponding format specifier. If you try to apply a string format specifier, like "%-7s", to a number, or a floating point specifier, like "%5.2f" to a string, you will get a Java error message.

Locale expression

The **USING\$()** function can localize the output string based on language and region. The locale specifies the language and region with standardized codes. The `<locale_sexp>` is a string containing zero or more codes separated by underscores.

The function accepts up to three codes. The first must be a language code, such as "en", "de", or "ja". The second must be a region or country code, such as "FR", "US", or "IN". Some language and country combinations can accept a third code, called the "variant code".

The function also accepts the standard three-letter codes and numeric codes for country or region. For example, "fr_FR", "fr_FRA", and "fr_250" are all equivalent.

If you want to use the default locale of your Android device, make the `<locale_exp>` an empty string (""), or leave it out altogether. If you leave it out, you must keep the comma: **USING\$(, "%f", x)**

If you make a mistake in the `<locale_sexp>`, you may get an obscure Java error message, but more likely your locale will be ignored, and your string will be formatted using the default locale of your device.

Android devices do not support all possible locales. If you specify a valid locale that your device does not understand, your string will be formatted using the default locale.

Format expression

If you are familiar with the *printf* functions of C/C++, Perl, or other languages, you will recognize most of format specifiers of this function. The format expression is exactly the same as format string of the Java

Formatter class, or the *format(String, Object...)* method of the Java *String*, with two exceptions: Boolean format specifiers are not supported, and hex hash specifiers are limited to numeric and string types.

If you have not programmed in one of those other languages, this will be your introduction to a powerful tool for formatting text.

A format expression is a string with embedded format specifiers. Anything that is not a format specifier is copied literally to the function output string. Each embedded format specifier is replaced with the value of an expression from the list, formatted according to the specifier. For example:

```
PRINT USING$("", "Pi is approximately %f.", PI()) ← function call
Pi is approximately 3.141593.                     ← printed output for English locale
```

The <locale_exp> is "", meaning "use my default locale".

The <format_exp>, "Pi is approximately %f", has one format specifier, "%f".

"%f" means, "use the default decimal floating point output format".

The expression list has one item, the math function **PI()**.

In the output, "%f" is replaced by the value of the the **PI()** function.

Your output may be different if your locale language is not English.

Format Specifiers

Here is a brief summary of the available format specifiers:

For this type of data	Use these formats	Comments
String	%s %S	%S forces output to upper-case
Number	%f %e %E %g %G %a %A	Standard BASIC! numbers are floating point Use %f for decimal output: "1234.567" Use %e or %E for exponential notation: "1.234e+03" %E writes upper-case: "1.234E+03" %g (%G) lets the system choose %f or %e (%E) %a and %A are "hexadecimal floating point"
Integer	%d %o %x %X	USING\$ can use some math functions as integers Use %d for decimal, %o for octal, %x %X for hex %x writes lower-case abcdef, %X writes upper-case
Special integer	%c %C %t	These specifiers can operate on an integer %c %C output a character, %C writes upper-case %t represents a family of time format specifiers
None	%% %n	These specifiers do not read the expression list %% writes a single "%" to the output %n writes a newline, exactly the same as \n

For more information about %a and %A, see the Android documentation linked above.

Android's %b and %B are not supported because BASIC! has no Boolean type.

Android's %h and %H hash code specifiers are limited to strings and numbers in BASIC!.

For an explanation of **USING\$()** with integer format specifiers, see below.

There is a whole family of time format specifiers: **%t<x>** where **<x>** is another letter. They operate on an integer, which they interpret as the number of milliseconds since the beginning of January 1, 1970, UTC (the "epoch"). You can apply time format specifiers to the output of the **TIME()** functions. Note, however, that the **%t** time specifiers use your local timezone, not the **TimeZone.set** value.

There are more than 30 time format specifiers. A few examples appear below, but to get the full list you should read the Android documentation linked above.

```
PRINT USING$("", "The time is: %tI:%<tM:%<tS %<Tp", time())    % the hard way
PRINT USING$("", "The time is: %tr", time())                  % same thing!
02:27:16 PM                                                    ← example of printed output

t = TIME(2001, 2, 3, 4, 5, 6)                                % set 2001/02/03 04:05:06, local timezone
PRINT USING$("sv", "%tA", int(t))                            % day in Swedish, prints "lördag"
PRINT USING$("es", "%tB", int(t))                            % month in Spanish, prints "febrero"
PRINT USING$("", "%tY/%tm/%td", int(t), int(t), int(t))      % prints "2001/02/03"
PRINT USING$( , "%tY/%<tm/%<td", int(t))                    % prints "2001/02/03"
PRINT USING$("en_GB", "%tH:%<tM:%<tS", int(t))              % prints "04:05:06"
PRINT USING$("in_IN", "%tT", int(t))                        % prints "04:05:06"
```

Note: Date and time are printed for your local timezone, regardless of either the **TIMEZONE.SET** setting or the locale parameter. Try the same set of examples with **TIMEZONE.SET "UTC"**. Unless that is your local timezone, a different hour and perhaps even a different day will be displayed.

Optional Modifiers

The format specifiers can be used exactly as shown in the table. They have default settings that control how they behave. You can control the settings yourself, fine-tuning the behavior to suit your needs.

You can modify the format specifiers with *index*, *flags*, *width*, and *precision*, as shown in this example:

%3\$,15.4f							
%	3\$	-,	15	.	4	f	"
	<index>	<flags>	<width>		<precision>	<specifier>	

Index

Normally the format specifiers are applied to the arguments in order. You can change the order with an argument index. An index a number followed by a **\$** character. The argument index **3\$** specifies the third argument in the list.

```
PRINT USING$("", "%3$s %s %s", "a", "b", "c")    % prints "c a b"
```

The special argument index **"<"** lets you reuse an argument.

```
PRINT USING$("", "%o %<d %<h", int(64) )        % prints "100 64 40"
```

In the last example, there is only one argument, but three format specifiers. This is not an error because the argument is reused.

Flags

There are six flags:

-	left-justify; if no flag, right-justify
+	always show sign; if no flag, show "-" but do not show "+"
0	pad numbers with leading zeros; if no flag pad with spaces
,	use grouping separators for large numbers
(put parentheses around negative values
#	alternate notation (leading 0 for octal, leading 0x for hexadecimal)

Width

The **width** control sets the minimum number of characters to print. If the value to format is longer than the width setting, the entire value is printed (unless it would exceed the **precision** setting). If the value to format is shorter than the **width** setting, it is padded to fill the width. By default, it is padded with spaces on the left, but you change this with the "-" and "0" flags.

Precision

The **precision** control means different things to different data types.

For floating point numbers, **precision** specifies the number of characters to print after the decimal point, padding with trailing zeros if needed.

For string values, it specifies the maximum number of characters to print. If **precision** is less than **width**, only **precision** characters are printed.

```
"%4s", "foo" → " foo"
"%-4s", "foo" → "foo "
"%4.2s", "foo" → "fo"
```

The **precision** control is not valid for other types.

In the example above, **%,15.4f**:

The **flags** "-" and "," mean "left-justify the output" and "use a thousands separator".

The **width** is 15, meaning the output is to be at least 15 characters wide.

The **precision** is 4, so there will be exactly four digits after the decimal point.

The whole format specifier means, "format a floating point number (%f) left-justified ("-") in a space 15 characters wide, with 4 characters after the decimal point, with a thousands separator (",")".

The characters used for the decimal point and the thousands separator depend on the locale:

```
"1,234.5678"    " for locale "en"
"1 234,5678"    " for locale "fr"
"1.234,5678"    " for locale "it"
```

Integer values

BASIC! has only double-precision floating point numbers. It does not have an integer type. The **USING\$()** function supports format specifiers ("%d", "%t", "%x") that apply only to integer values.

USING\$() has a special relationship with the math functions that intrinsically produce integer results. BASIC! converts the output of these functions to floating point, for storage in numeric variables, but **USING\$()** can get the original integer values. For example:

```
PRINT USING$("", "%d", 123)           % ERROR!
PRINT USING$("", "%d", INT(123))      % No error
```

The functions that can produce integer values for **USING\$()** are:

```
INT()    BIN()    OCT()    HEX()
CEIL()   FLOOR()
ASCII()  UCODE()
BAND()   BOR()    BXOR()
SHIFT()  TIME()
```

FORMAT_USING\$(<locale_sexp>, <format_sexp> { , <exp>}...)

Alias for **USING\$()**. You can use the two equivalent functions to make your code easier to read. For example:

```
string$ = FORMAT_USING("", "pi is not %d", int(pi()))
Print USING$("en_US", "Balance: $$8.2f", balance)
```

FORMAT\$(<pattern_sexp>, <nexp>)

Returns a string with <nexp> formatted by the pattern <pattern_sexp>.

Leading Sign	A negative (-) character for numbers < 0 or a space for numbers >= 0. The Sign and the Floating Character together form the Floating Field .
Floating Character	If the first character of the pattern is not "#" or "." or "-" then that character becomes a "floating" character. This pattern character is typically a "\$". If no floating character is provided then a space character is used. See also Overflow , below.
Decimal Point	The pattern may have one optional decimal point character ("."). If the pattern has no decimal point, then only the whole number is output. Any digits that would otherwise appear after the decimal point are not output.
# Character (before decimal, or no decimal)	Each "#" is replaced by a digit from the number. If there are more "#" characters than digits, then the leading "#" character(s) are replaced by space(s) .
# Character (after decimal point)	Each "#" is replaced by a digit from the number. If there are more "#" characters than significant digits, then the trailing "#" character(s) are replaced by zero(s) . The number of "#" characters after the pattern decimal point specifies the number of decimal digits that will be output.
% Character (before decimal, or no decimal)	Each "%" is replaced by a digit from the number. If there are more "%" characters than digits, then the leading "%" character(s) are replaced by zero(s) .

% Character (after decimal)	The "%" character is not allowed after the decimal point. This is a syntax error.
Non-pattern Characters	If any pattern character (other than the first) is not # or %, then that character is copied directly into the output. If the character would appear before the first digit of the number, it is replaced by a space. This feature is usually used for commas.
Overflow	If the number of digits exceeds the number of # and % characters, then the output has the ** characters inserted in place of the Floating Field.
Output Size	The number of characters output is always the number of characters in the pattern plus one for the sign plus one more for the space if the pattern has no Floating Character.

Notes

The sign and the floating character together form a **Floating Field** two characters wide that always appears just before the first digit of the formatted output. If there are any leading spaces in the formatted output, they are placed before the floating field.

The "#" character generates leading spaces, not leading zeros. "###.###" formats 0.123 as ".123". If you want a leading zero, use a "%". For example "%.###", "%#.###", or "##%" all assure a leading zero.

Be careful mixing # and % characters. Doing so except as just shown can produce unexpected results.

The number of characters output is always the number of characters in the pattern plus the two floating characters.

Examples

Function Call	Output	Width
Format\$("###,###,###", 1234567)	1,234,567	12 characters
Format\$("%%,%%,%%,%#.", 1234567.89)	01,234,567.8	14 characters
Format\$("\$###,###", 123456)	\$123,456	9 characters
Format\$("\$###,###", -1234)	-\$1,234	9 characters
Format\$("\$###,###", 12)	\$12	9 characters
Format\$("\$%%,%%,%#.", -12)	-\$000,012	9 characters
Format\$("##.#", 0)	.0	6 characters
Format\$("%#.#", 0)	0.0	6 characters
Format\$("\$###.##", -1234.5)	**234.50	8 characters

User-Defined Functions

User-Defined Functions are BASIC! functions like ABS(n), MOD(a,b) and LEFT\$(a\$,n) except that the operation of the function is defined by the user. User functions should generally be defined at the start of the program and in particular, they should appear before the places where they are called.

User-Defined Functions may call other User-Defined Functions. A function can even recursively call itself.

You may define a function with the same name as a built-in function. The User-Defined Function always overrides the built-in function, and the built-in function is not accessible.

Each time a function is called from another function a certain amount of memory is used for the execution stack. The depth of these nested calls is limited only by the amount of memory that your particular Android device allocates to applications.

Variable Scope

All variables created while running a User-Defined Function are private to the function. A variable named `v$` in the main program is not the same as variable `v$` within a function. Furthermore, a variable named `v$` in a recursively called function is not the same `v$` in the calling function.

A function cannot access variables created outside of the function, except as parameters passed by reference. (See **`Fn.def`**, below, for an explanation of parameters passed by value and by reference.)

All variables created while running a User-Defined Function are destroyed when the function returns. When an array variable is destroyed, its storage is reclaimed. However, when a data structure pointer is destroyed, the data structure is not destroyed (see next section).

Data Structures in User-Defined Functions

Data structures (List, Stack, Bundle, bitmap, graphical object – anything referenced through a pointer) are global in scope. That is, if a variable is used as a pointer to a data structure, it points to the same data structure whether it is used inside or outside of a function. The data structure may have been created in the main program, the same user-defined function, or some other user-defined function.

This means that if you pass a pointer to a bundle, for example, and modify that bundle inside the function, the changes will be retained when the function returns. It also means that a function can modify graphical objects created outside of the function.

Data structures (List, Stack, Bundle, or graphical object) created while running a User-Defined Function are not destroyed when the function returns. Local variables that point to the data structures are lost, but you can return a data structure pointer as the function's return value or through a parameter passed by reference.

Commands

**`Fn.def name|name$({nvar}|{svar}|Array[]|Array$[], ...
{nvar}|{svar}|Array[]|Array$[])`**

Begins the definition of a function. This command names the function and lists the parameters, if any.

If the function name ends with the `$` character then the function will return a string, otherwise it will return a number. The parameter list can contain as many parameters as needed, or none at all. The parameters may be numeric or string, scalar or array.

Your program must execute **`Fn.def`** before it tries to call the named function. Your program must not attempt to create more than one function with the same name, or the same function more than once. However, you may override a built-in function by defining your own function with the same name.

The following are all valid:

```
FN.DEF cut$(a$, left, right)
FN.DEF sum(a, b, c, d, e, f, g, h, i, j)
FN.DEF sort(v$[], direction)
FN.DEF pi() % Overrides built-in. You can make  $\pi = 3!$ 
```

Parameters create variables visible only inside the function. They can be used like other variables created inside the function (see Variable Scope, above).

There are two types of parameters: call by reference and call by value. Call by value means that the calling variable value (or expression) is copied into the called variable. Changes made to the called variable within the function do not affect the value of the calling variable. Call by reference means that the calling variable value is changed if the called variable value is changed within the function.

Scalar (non-array) function variables can be either call by value or call by reference. Which type the variable will be depends upon how it is called. If the calling variable has the "&" character in front of it, then the variable is call by reference. If there is no "&" in front of the calling variable name then the variable is call by value.

```
FN.DEF test(a)
a = 9
FN.RTN a
FN.END

a = 1
PRINT test(a), a %will print: 9, 1
PRINT test(&a), a %will print: 9, 9
```

Array parameters are always call by reference.

```
FN.DEF test(a[])
a[1] = 9
FN.RTN a[1]
FN.END

DIM a[1]
a[1] = 1
PRINT test(a[]), a[1] % prints: 9, 9
```

Along with the function's return value, you can use parameters passed by reference to return information to a function's caller.

Fn.rtn <sexp>|<nexp>

Causes the function to terminate execution and return the value of the return expression <sexp>|<nexp>. The return expression type, string or number, must match the type of the function name. **Fn.rtn** statements may appear anywhere in the program that they are needed.

A function can return only a single scalar value. It cannot return an array. It cannot return a data structure (List, Stack, Bundle, or graphical object), but it can return a pointer to a data structure.

Note: You can also return information to a function's caller through parameters passed by reference.

Fn.end

Ends the definition of a user-defined function. Every function definition must end with **Fn.end**.

When your function is running, executing the **Fn.end** statement causes the function to terminate and return a default value. If the function type is numeric then the default return value is 0.0. A string function returns the empty string ("").

Call <user_defined_function>

Executes the user-defined function. Any value returned by the function will be discarded.

The **CALL** command keyword is optional. Just as BASIC! can infer the **LET** command from a line that starts with a variable, it can infer the **CALL** command from a line that starts with a function name.

For example, if you have defined a function like this:

```
FN.DEF MyFunction(x, y$, z)
  < your code here >
FN.END
```

You can execute the function, ignoring its return value, with either of these statements:

```
CALL MyFunction(a, b$, c)
MyFunction(a, b$, c)
```

As with **LET**, you must use **CALL** if your function name starts with a BASIC! command keyword. It is also a little faster to execute a function with **CALL** than to make BASIC! infer the command. See **LET**, above, for details.

Program Control Commands

If / Then / Else / Elseif / Endif

The **If** commands provide for the conditional execution of blocks of statements. (Note: the braces { } are not part of the command syntax. They are used only to show parts that are optional.)

```
IF <condition> { THEN }
  <statement>
  <statement>
...
  <statement>
{ ELSEIF<condition> { THEN }
  <statement>
  <statement>
```

```

...
    <statement> }
{ ELSE
    <statement>
    <statement>
...
    <statement> }
ENDIF

```

If commands may be nested to any depth. That is, any <statement> in a block may be a full **If** command with all of its own <statement> blocks.

See the Sample Program file, F04_if_else.bas, for working examples of the **If** command.

If / Then / Else

If your conditional block(s) contain(s) only one statement, you may use a simpler form of the **If** command, all one line:

```
IF <condition> THEN <statement> { ELSE <statement> }
```

In this form, **Then** is required, and there is no **Elseif** or **Endif**.

This form does not nest: neither <statement> may be an **If** command.

Because the single statements are not treated as blocks, this is the preferred form if either of the embedded statements is a **Break**, **Continue**, or **GoTo**.

You may replace either <statement> with multiple statements separated by colon (":") characters. If you do this, the set of multiple statements is treated as a block, and the single-line **If/Then/Else** becomes an **If/Then/Else/Endif**. These two lines are exactly equivalent:

```
IF (x > y) THEN x = y : PRINT a$ ELSE y = x : PRINT b$
IF (x > y) : x = y : PRINT a$ : ELSE : y = x : PRINT b$ : ENDIF

```

Please note, if you wish to use colon-separated statements in this form of **If/Then/Else**, then you must be careful to put spaces around the keywords **Then** and **Else**. Spaces are not significant to the BASIC! interpreter, but they are needed by the preprocessor that converts the single-line **If** with multi-statement blocks into a multi-line **If** with an **Endif**.

For - To - Step / Next

```

FOR <nvar> = <nexp_1> TO <nexp_2> {STEP <nexp_3>}
    <statement>
    ...
    <statement>
NEXT {<nvar>}

```

Initially, <nvar> is assigned the value of <nexp_1> and compared to <nexp_2>. {STEP <nexp_3>} is optional and may be omitted. If omitted then the **Step** value is 1.

If <nexp_3> is positive then

if <nvar> <= <nexp_2> then
the statements between the **For** and **Next** are executed.

If <nexp_3> is negative then

if <nvar> >= <nexp_2> then
the statements between the **For** and **Next** are executed.

When the **Next** statement is executed, <nvar> is incremented or decremented by the **Step** value and the test is repeated. The <statement>s will be executed as long as the test is true. Each time, <nvar> is compared to the original value of <nexp_2>; <nexp_2> is not re-evaluated with each **Next**.

Because the keywords **To** and **Step** are in the middle of the line with expressions that may include variables, it is possible to confuse BASIC!. Remember that the interpreter does not see any spaces you put between variables and keywords. **FOR a TO m** is seen as **foratom**. If there is any possibility of confusion, use parentheses to tell BASIC! that a name is a variable:

FOR WinTop TO WinBot	% ERROR: interpreted as "FOR win TO ptowinbot"
FOR (WinTop) TO WinBot	% interpreted as intended

For-Next loops can be nested to any level. When **For-Next** loops are nested, any executed **Next** statement will apply to the currently executing **For** statement. This is true no matter what the <nvar> coded with the **Next** is. For all practical purposes, the <nvar> coded with the **Next** should be considered to be nothing more than a comment.

It is possible to exit a **For** loop without **Next** or **F_N.break**. However, this can create subtle logic errors that are hard to debug. If you set debug mode (see the **Debug.on** command), then BASIC! can help you find these bugs. When your program ends and debug is on, if your program entered a **For** loop and did not leave it cleanly, BASIC! shows a run-time error: "Program ended with FOR without NEXT".

F_N.continue

If this statement is executed within a **For-Next** loop, the rest of the current pass of the loop is skipped. The **Next** statement executes immediately.

F_N.break

If this statement is executed within a **For-Next** loop, the rest of the current pass of the loop is skipped and the loop is terminated. The statement immediately following the **Next** will be executed.

While <lexp> / Repeat

```
While <lexp>  
  <statement>
```



```

    ...
    <statement>
Repeat

```

The <statement>s between the **While** and **Repeat** will be executed as long as <lexp> evaluates as true. The <statements>s will not be executed at all if <lexp> starts off false.

While-Repeat loops may be nested to any level. When **While-Repeat** are nested, any executed **Repeat** statement will apply to inner most **While** loop.

W_R.continue

If this statement is executed within a **While-Repeat** loop, the rest of the current pass of the loop is skipped. The **Repeat** statement executes immediately.

You can exit a **While** loop without **Repeat** or **W_R.break**. As with **For-Next** loops, this can create subtle bugs, and BASIC! can help you find them. If debug is on, and your program is still in a **While** loop when it ends, BASIC! shows a run-time error: "Program ended with WHILE without REPEAT".

W_R.break

If this statement is executed within a **While-Repeat** loop, the rest of the current pass of the loop is skipped and the loop is terminated. The statement immediately following the **Repeat** will be executed.

Do / Until <lexp>

```

Do
    <statement>
    ...
    <statement>
Until <lexp>

```

The statements between **Do** and **Until** will be executed until <lexp> is true. The <statement>s will always be executed at least once.

Do-Until loops may be nested to any level. Any encountered **Until** statement will apply to the last executed **DO** statement.

You can exit a **Do** loop without **Until** or **D_U.break**. As with **For-Next** loops, this can create subtle bugs, and BASIC! can help you find them. If debug is on, and your program is still in a **Do** loop when it ends, BASIC! shows a run-time error: "Program ended with DO without UNTIL".

D_U.continue

If this statement is executed within a **Do-Until** loop, the rest of the current pass of the loop is skipped. The **Until** statement executes immediately.

D_U.break

If this statement is executed within a **Do-Until** loop, the rest of the current pass of the loop is skipped and the loop is terminated. The statement immediately following the **Until** will be executed.

Labels, GOTO, GOSUB, and RETURN: Traditional BASIC

Before computer scientists invented the looping structures and user-defined functions described above, program flow was controlled by **GOTO**, **GOSUB**, and **RETURN** statements. These statements are present in BASIC!, too, mainly for compatibility with old BASIC dialects.

A **GoTo** statement is a one-way jump to another place in your program, identified by a **Label**. The program goes to the **Label** and continues execution there.

A **GoSub** is similar, except that the **Label** is the beginning of a "Subroutine". The program goes to the **Label**, and executes there until it reaches a **Return** statement. Then it "returns", going back to where it came from: the line after the **GoSub** statement.

Extensive use of the **GoTo** command in your program should be generally avoided. It can make code hard to read and harder to debug. Instead, you should use structured elements like **Do...Until**, **While...Repeat**, etc. in conjunction with the **Break** and **Continue** statements.

It is especially serious to use **GoTo** commands inside an **If...Else...Endif**, **For...Next**, or other structured block, jumping to code outside of the block. Doing this consumes system resources and may corrupt BASIC!'s internal data structures. This practice may lead your program to a run-time error:

```
Stack overflow. See manual about use of GOTO.
```

Label

A label is a word followed by the colon ":", character. Label names follow the same conventions as variable names, except that a label must not start with a BASIC! command keyword.

You may put a label on a line with other commands. Use two colons: one to signify that the word is a label, and a second to separate the label from the other command(s) on the line.

```
Here: : IF ++a < 5 THEN GOTO here ELSE PRINT a
```

This program prints 5.0 and then stops.

The colon signifies that the word is a label, but it is not part of the label. Use the colon where the label is defined. Do not use it in the **GoTo** or **GoSub** that jumps to the label.

For example:

```
This_is_a_Label:
@Label#3:
Loop:          % The command "GoTo Loop" jumps to this line
               <statement>
```

```
...  
<statement>  
GoTo Loop
```

GoTo <label>

The next statement that will be executed will be the statement following <label>.

GoTo <index_nexp>, <label>...

A "computed GOTO". The index expression is evaluated, rounded to the nearest integer, and used as an index into the list of labels. The program jumps to the statement after the indexed label. If the index does not select any label, the program continues at the statement after the **GoTo**.

For examples, see **GoSub**.

GoSub <label> / Return

The next statement that will be executed will be the statement following <label>.

The statements following the line beginning with <label> will continue to be executed until a **Return** statement is encountered. Execution will then continue at the statement following the **GoSub** statement.

Example:

```
Message$ = "Have a good day"  
GOSUB xPrint  
PRINT "Thank you"  
<statement>  
...  
<statement>  
END  
xPrint:  
PRINT Message$  
RETURN
```

This will print:

```
Have a good day  
Thank you
```

GoSub <index_nexp>, <label>... / Return

A "computed GOSUB". The index expression is evaluated, rounded to the nearest integer, and used as an index into the list of labels. The program jumps to the statement after the indexed label. When the next **Return** instruction executes, the program returns to the statement after this **GoSub**.

If the index does not select any label, the program continues to the statement after the **GoSub**. No subroutine is executed, and no **Return** statement is expected.

Example:

```

d = FLOOR(6 * RND() + 1)           % roll a six-sided die
GOSUB d, Side1, Side2, Side3, Side4, Side5, Side6
PRINT "Welcome back!"
END
Side1:
<subroutine for side 1>
RETURN
...
Side6:
<subroutine for side 6>
RETURN

```

Using Source Code from Multiple Files

Include FilePath

Before the program is run, the BASIC! preprocessor replaces any **Include** statements with the text from the named file. You can use this to insert another BASIC! program file into your program at this point. The program is not yet running, therefore the File Path cannot be a string expression.

You may include a file only once. If multiple **Include** statements name the same file, the preprocessor inserts the contents of the file in place of the first such **Include** statement and deletes the others. This prevents Out-Of-Memory crashes caused by an **Include** file including itself.

```
INCLUDE functions/DrawGraph.bas
```

inserts the code from the file "<pref base drive>/rfo-basic/source/functions/DrawGraph.bas" into the program.

The File Path may be written without quotation marks, as in the example above, or with quotes:

```
INCLUDE "functions/DrawGraph.bas"
```

If present, the quotes prevent the preprocessor from forcing the File Path to lower-case. Normally, this does not change how BASIC! behaves, because the file system on the SD card is case-insensitive.

DrawGraph.bas and **drawgraph.bas** both refer to the same file.

However, if you build your program into a standalone Android application (see **Appendix D**), you can use virtual files in the Android **assets** file system. File names in **assets** are case-sensitive, so you may need to use quotes with the **Include** File Path.

Because **Include** is processed before your program starts running, it is not affected by program logic.

```

IF 0
  INCLUDE functions/DrawGraph.bas
ENDIF

```

In this example, the contents of the program file replace the **Include** statement in the body of the **If/Endif**, but the statements are never executed because **If 0** is always false.

However, an **Include** in a single-line **If** is ignored:

```
IF x THEN INCLUDE functions/DrawGraph.bas ELSE PRINT "bad!"
```

In this example, the file is not inserted in place of the **Include** statement.

Run <filename_sexp>{<data_sexp>}

This command will terminate the running of the current program and then load and run the BASIC! program named in the filename string expression. The filename is relative to BASIC's "source/" directory. If the filename is "program.bas" and your <pref base drive> is "/sdcard" (the default), then the file "/sdcard/rfo-basic/source/program.bas" will be executed.

If the filename parameter is omitted, and the currently executing program has a name, then the program restarts. The program does not have a name if you run from the BASIC! Editor without first saving the program; in this case **Run** terminates your program with a syntax error.

The optional data string expression provides for the passing of data to the next program. The passed data can be accessed in the next program by referencing the special variable, **##\$**.

Run programs can be chained. A program loaded and run by means of the **Run** command can also run another program file. This chain can be as long as needed.

When the last program in a **Run** chain ends, tapping the BACK key will display the original program in the BASIC! Editor.

When a program ends with an error, the Editor tries to highlight the line where the error occurred. If the program with the error was started by a **Run** command, the Editor does not have that program loaded. Any highlighting that may be displayed is meaningless.

Program.info <nexp>|<nvar>

Returns a Bundle that reports information about the currently running program. If you provide a variable that is not a valid Bundle pointer, the command creates a new Bundle and returns the Bundle pointer in your variable. Otherwise it writes into the Bundle your variable or expression points to.

The bundle keys and possible values are in the table below:

Key	Type	Value
BasPath	String	Full path + name of the program currently being executed. The path is relative to BASIC!'s "source/" directory.
BasName	String	Name of the program currently being executed.
SysPath	String	Full path to the BASIC!'s private file storage directory. The path is relative to BASIC!'s "data/" directory.
UserApk	Numeric (Logical)	Returns 1.0 (true) if the current program is being run from a standalone user-built APK (Appendix D). Returns 0.0 (false) if the program is being from from the BASIC! Editor or a Launcher Shortcut (Appendix C).

For example, assume:

- You are using the default <pref base drive>
- You downloaded a file called "my_program.bas" to the standard Android Download directory.
- You used the BASIC! Editor to load and run the downloaded program.

Then the returned values would be as follows:

Key	Value	Key	Value
BasPath	.././Download/my_program.bas	BasName	my_program.bas
SysPath	.././../././data/data/com.rfo.basic	UserApk	0.0

SysPath: BASIC! normally keeps programs and data in its *base directory* (see **Working with Files**, later in this manual). The base directory is in public storage space. BASIC! programs also have access to a private storage area. Your program can create a subdirectory within the SysPath directory and store private files there. Note that if you uninstall BASIC!, any files in private storage will be deleted.

SysPath is of particular interest to you if you build a BASIC! program as an application in a standalone apk, as described in Appendix D. See **Modifications to the AndroidManifest.xml File → Permissions**.

Switch Commands

The Switch commands may be used to replace nested if-then-else operations.

```
SW.BEGIN a
SW.CASE 1
    <statement1>
    ...
    <statement2>
    SW.BREAK
SW.CASE 2, 4
    <statement3>
    ...
    <statement4>
    SW.BREAK
SW.CASE < 0
    <statement5>
    ...
    <statement6>
    SW.BREAK
SW.DEFAULT
    <statement7>
SW.END
```

The value of the argument of **Sw.begin** is compared to the argument of each **Sw.case** in order. If any **Sw.case** matches the **Sw.begin**, the statements following the matching **Sw.case** is executed; if no **Sw.case** matches, the statements after **Sw.default** are executed. Once BASIC! starts to execute the statements of a **Sw.case** or **Sw.default**, it continues execution until it finds a **Sw.break** or the **Sw.end**, ignoring any other **Sw.case** or **Sw.default** it may encounter. **Sw.break** causes a jump to the **Sw.end**.

In the example:

- if the value of **a** is 1, then <statement1> through <statement2> execute
- if the value of **a** is 2 or 4, then <statement3> through <statement4> execute
- if the value of **a** is less than 0, then <statement5> through <statement6> execute
- if **a** has any other value (3, or more than 4) then <statement7> executes.

A **Sw.begin** must be followed by a **Sw.end**, and all of the **Sw.case** and the **Sw.default** (if there is one) for the same switch must appear between them. A set of switch commands is treated as a single unit, just as if BASIC! were compiled.

Nesting Switch Operations

Switches can be nested. The block of statements following a **Sw.case** or **Sw.default** may include a full set of switch commands. The nested switch begins with another **Sw.begin** and ends with another **Sw.end**, with its own **Sw.case** and **Sw.default** statements in between. You can nest other switches inside nested switches, for as many levels as you want.

If you prefer, you may put the inner switch operations in a labeled **Gosub** routine or a User-defined Function, and put the **Gosub** or function call in the **Sw.case** or **Sw.default** block. This may make your code easier to read, and it will also make the initial scan of the switch a little faster.

Sw.begin <exp>

Begins a switch operation. BASIC! scans forward until it reaches a **Sw.end**, locating all **Sw.case**, **Sw.break**, and **Sw.default** statements between the **Sw.begin** and the **Sw.end**.

The numeric or string expression <exp> is evaluated. Its value is then compared to the expression(s) in each **Sw.case** statement, in order. BASIC! uses the result of the compares to decide which statement to execute next.

IF this condition is met:	THEN jump to the statement
One or more Sw.Case statement(s) match the Sw.Begin	after the <i>first</i> matching Sw.case .
No Sw.Case matches AND a Sw.default exists	after the Sw.default .
No Sw.Case matches AND no Sw.default exists	after the Sw.end .

There are two forms of **Sw.case**. You may freely mix both forms. Each form defines what it means to "match" **Sw.begin**. Only the first matching **Sw.case** has any effect.

Sw.case <exp>, ...

This form of **Sw.case** provides a list of one or more expressions. A **Sw.case** of this form matches the **Sw.begin** if the value of at least one of the expressions *exactly equals* the value of the **Sw.begin** parameter. The type of the **Sw.begin** and **Sw.case** parameter(s), numeric or string, must match.

Sw.case <op><exp>

The first form of **Sw.case**, above, matches the **Sw.begin** if the value of any of its parameters *equals* the value of the **Sw.begin** parameter. This second form can take only one expression <exp>, but it lets you specify a different logical operator <op>. You may use any of these comparison operators:

< <= > >= <>

For example:

```
SW.BEGIN a
SW.CASE < b           % This SW.CASE matches if a < b
```

The expression <exp> may be arbitrarily complex. The whole expression is evaluated as if written:

<value of Sw.begin argument> <op> <exp>

Operator precedence is applied as usual.

Sw.break

This statement may be used to terminate the block of statements that follows **Sw.case** or **Sw.default**. The **Sw.break** causes BASIC! to jump forward to the **Sw.end** statement, skipping everything between.

If no **Sw.break** is present in a particular **Sw.case** then subsequent **Sw.cases** will be executed until a **Sw.break** or **Sw.end** is encountered.

Sw.default

This statements acts like a **Sw.case** that matches any value. If any **Sw.case** matches the **Sw.begin** value, then the **Sw.default** is ignored, even if the matching **Sw.case** is after the **Sw.default**.

A switch is not required to have a **Sw.default**, but it must not have more than one. A second **Sw.default** in the same switch is a syntax error.

Sw.end

The **Sw.end** terminates a switch operation. **Sw.end** must eventually follow a **Sw.begin**.

Interrupt Labels (Event Handlers)

You can perform physical actions that tell your BASIC! program to do something. When you touch the screen or press a key you cause an *event*. These events are *asynchronous*, that is, they happen at times your program cannot predict. BASIC! detects some events so your program can respond to them.

BASIC! handles events as *interrupts*. Each event that BASIC! recognizes has a unique *Interrupt Label*. When an event occurs, BASIC! looks for the Interrupt Label that matches the event.

- If you have not written that Interrupt Label into your program, the event is ignored and your program goes on running as if nothing happened.
- If you have included the right Interrupt Label for the event, BASIC! jumps to that label and continues execution at the line after the label. This is called trapping the event.

BASIC! does not necessarily respond to the event as soon as it occurs. The statement that is executing when the event occurs is allowed to complete, then BASIC! jumps to the Interrupt Label.

When you use an Interrupt Label to trap an event, BASIC! executes instructions until it finds a *Resume* command that matches the Interrupt Label. During that time, it records other events but it does not respond to them. The block of code between the Interrupt Label and the matching Resume may be called an *Interrupt Service Routine (ISR)* or, if you prefer, an *Event Handler*.

When BASIC! executes the event's Resume command, it resumes normal execution.

- BASIC! jumps back to where it was running when the interrupt occurred.
- BASIC! again responds to other events, including any that occurred while it was handling an event.

An Interrupt Label looks and behaves just like any other label in BASIC!. However, you must not execute any of the Resume commands except to finish an event's handler.

All Interrupt Labels

BASIC! supports trapping of the following events. These interrupt labels and their Resume commands are described in various parts of this manual.

- **OnBackground:** **Background.resume**
- **OnBackKey:** **Back.resume**
- **OnBtReadReady:** **Bt.onReadReady.resume**
- **OnConsoleTouch:** **ConsoleTouch.resume**
- **OnError:** None (not a true event, see **OnError:**, below)
- **OnGrTouch:** **Gr.onGrTouch.resume**
- **OnKbChange:** **Kb.resume**
- **OnKeyPress:** **Key.resume**
- **OnLowMemory:** **LowMemory.resume**
- **OnMenuKey:** **MenuKey.resume**
- **OnTimer:** **Timer.resume**

OnError:

Special interrupt label that traps a run-time error as if it were an event, except that:

- **OnError:** has no matching Resume command. You can use **GoTo** to jump anywhere in your program.
- **OnError:** is not locked out by other interrupts.
- **OnError:** does not lock out other interrupts.

If a BASIC! program does not have an **OnError:** label and an error occurs while the program is running, an error message is printed to the Output Console and the program stops running.

If the program does have an **OnError:** label, BASIC! does not stop on an error. Instead, it jumps to the **OnError:** label (see "Interrupt Labels", above). The error message is not printed, but it can be retrieved by the **GETERROR\$()** function.

Be careful. An infinite loop will occur if a run-time error occurs within the **OnError:** code. You should not place an **OnError:** label into your program until the program is fully debugged. Premature use of **OnError:** will make the program difficult to debug.

OnConsoleTouch:

Interrupt label that traps a tap on a line printed on the Output Console. BASIC! executes the statements following the **OnConsoleTouch:** label until it reaches a **ConsoleTouch.resume**. Note that you must touch a line written by a **PRINT** command, although it may be blank. BASIC! ignores any touch in the empty area of the screen below the printed lines.

After touching a Console line, you may use the **Console.Line.Touched** command to determine what line of text was touched.

This label allows the user to interrupt an executing BASIC! program in Console mode (not in Graphics mode). A common reason for such an interrupt would be to have the program request input via an **INPUT** statement. See the Sample File, **f35_bluetooth.bas**, for an example of this.

To detect screen touches while in graphics mode, use **OnGrTouch:**.

ConsoleTouch.resume

Resumes execution at the point in the BASIC! program where the **OnConsoleTouch:** interrupt occurred.

OnBackKey:

Interrupt label that traps the BACK key. BASIC! executes the statements following the **OnBackKey:** label until it reaches a **Back.resume**. If a BASIC! program does not have an **OnBackKey:** label, the BACK key normally halts program execution.

If you trap the BACK key with **OnBackKey:**, the BACK key does not stop your program. You should either terminate the run in the **OnBackKey:** code or provide another way for the user to tell your program to stop, especially if the program is in Graphics mode where there is no menu. If you do not then there will be no stopping the program (other than using Android Settings or a task killer application).

Back.resume

Resumes execution at the point in the BASIC! program where the **OnBackKey:** interrupt occurred.

OnMenuKey:

Interrupt label that traps the MENU key. BASIC! executes the statements following the **OnMenuKey:** label until it reaches a **MenuKey.resume**. Note: This interrupt does not work unless your device has a MENU key. Android devices since Honeycomb typically do not have a MENU key.

MenuKey.resume

Resumes execution at the point in the BASIC! program where the **OnMenuKey:** interrupt occurred.

OnKeyPress:

Interrupt label that traps a tap on any key. BASIC! executes the statements following the **OnKeyPress:** label until it reaches a **Key.resume**.

Key.resume

Resumes execution at the point in the BASIC! program where the **OnKeyPress:** interrupt occurred.

OnLowMemory:

Interrupt label that traps the Android "low memory" warning. BASIC! executes the program lines between the **OnLowMemory:** interrupt label and the **LowMemory.resume** command.

If Android is running out of memory, it may kill applications running in the background, but first it will broadcast a "low memory" warning to all applications. If you do not have an **OnLowMemory:** label in your program, you will see "Warning: Low Memory" printed on the Console.

LowMemory.resume

Resumes execution at the point in the BASIC! program where the **OnLowMemory:** interrupt occurred.

End{ <msg_sexp>}

Prints a message and stops the execution of the program. The default message is "END". You can use the optional <msg_sexp> argument to specify a different message. The empty string ("") prints nothing, not even a blank line. The **End** statement always stops execution, even if the statement has an error.

End statements may be placed anywhere in the program.

Exit

Causes BASIC! to stop running and exit to the Android home screen.

READ – DATA – RESTORE Commands

These commands approximate the **READ**, **DATA** and **RESTORE** commands of Dartmouth Basic.

Read.data <number>|<string>{<number>|<string>...,<number>|<string>}

Provides the data value(s) to be read with **Read.next**.

Read.data statements may appear anywhere in the program. You may have as many **Read.data** statements as you need.

Example:

```
Read.data 1,2,3,"a","b","c"
```

Read.data is equivalent to the **DATA** statement in Dartmouth Basic.

Read.next <var>, ...

Reads the data pointed to by the internal NEXT pointer into the next variables. The NEXT pointer is initialized to "1" and is incremented by one each time a new value is read. Data values are read in the sequence in which they appeared in the program **Read.data** statement(s).

The data type (number or string) of the variable must match the data type pointed by the NEXT pointer.

Example:

```
Read.next a,b,c,c$  
Read.next d$,e$
```

Read.next is equivalent to the **READ** statement in Dartmouth Basic

Read.from <nexp>

Sets the internal NEXT pointer to the value of the expression. This command can be set to randomly access the data.

The command **Read.from 1** is equivalent to the **RESTORE** command in Dartmouth Basic.

Debug Commands

The debug commands help you debug your program. The **Debug.on** command controls execution of all the debug commands. The debug commands are ignored unless the **Debug.on** command has been previously executed. This means that you can leave all your debug commands in your program and be assured that they will not execute unless you turn debugging on with **Debug.on**.

Debug.on

Turns on debug mode. All debug commands will be executed when in the debug mode.

Debug.on also enables a simple debugging aid built into BASIC!. If debug is on, and your program entered a loop but did not exit the loop cleanly, you will get a run-time error. See the looping commands (**For**, **While**, and **Do**) for details.

Debug.off

Turns off debug mode. All debug commands (except **Debug.on**) will be ignored. When your program exits, the broken-loop checks are not performed.

Debug.echo.on

Turns on Echo mode. Each line of the running BASIC! program is printed before it is executed. This can be of great help in debugging. The last few lines executed are usually the cause of program problems. The Echo mode is turned off by either the **Debug.echo.off** or the **Debug.off** commands.

Echo.on

Same as **Debug.echo.on**.

Debug.echo.off

Turns off the Echo mode.

Echo.off

Same as **Debug.echo.off**.

Debug.print

This command is exactly the same as the **Print** command except that the print will occur only while in the debug mode.

Debug.dump.scalars

Prints a list of all the Scalar variable names and values. Scalar variables are the variable names that are not Arrays or Functions. Among other things, this command will help expose misspelled variable names.

Debug.dump.array Array[]

Dumps the contents of the specified array. If the array is multidimensional the entire array will be dumped in a linear fashion.

Debug.dump.bundle <bundlePtr_nexp>

Dumps the Bundle pointed to by the Bundle Pointer numeric expression.

Debug.dump.list <listPtr_nexp>

Dumps the List pointed to by the List Pointer numeric expression.

Debug.dump.stack <stackPtr_nexp>

Dumps the Stack pointed to by the Stack Pointer numeric expression.

Debug.show.scalars

Pauses the execution of the program and displays a dialog box. The dialog box prints a list of all the Scalar variable names and values, the line number of the program line just executed and the text of that line. Scalar variables are the variable names that are not Arrays or Functions. Among other things, this command will help expose misspelled variable names.

For a description of the dialog box controls, see the **Debug.show** command, below.

Debug.show.array Array[]

Pauses the execution of the program and displays a dialog box. The dialog box prints the contents of the specified array, the line number of the program line just executed and the text of that line. If the array is multidimensional the entire array will be displayed in a linear fashion.

For a description of the dialog box controls, see the **Debug.show** command, below.

Debug.show.bundle <bundlePtr_nexp>

Pauses the execution of the program and displays a dialog box. The dialog box prints the Bundle pointed to by the Bundle Pointer numeric expression, the line number of the program line just executed and the text of that line.

For a description of the dialog box controls, see the **Debug.show** command, below.

Debug.show.list <listPtr_nexp>

Pauses the execution of the program and displays a dialog box. The dialog box prints the List pointed to by the List Pointer numeric expression, the line number of the program line just executed and the text of that line.

For a description of the dialog box controls, see the **Debug.show** command, below.

Debug.show.stack <stackPtr_nexp>

Pauses the execution of the program and displays a dialog box. The dialog box prints the Stack pointed to by the Stack Pointer numeric expression, the line number of the program line just executed and the text of that line.

For a description of the dialog box controls, see the **Debug.show** command, below.

Debug.watch var, ...

Gives a list of Scalar variables (not arrays) to be watched. The values of these variables will be shown when the Debug.show.watch command is executed. This command is accumulative, meaning that subsequent calls will add new variables into the watch list.

Debug.show.watch

Pauses the execution of the program and displays a dialog box. The dialog box lists the values of the variables being watched, the line number of the program line just executed and the text of that line.

For a description of the dialog box controls, see the **Debug.show** command, below.

Debug.show.program

Pauses the execution of the program and displays a dialog box. The dialog box shows the entire program, with line numbers, as well as a marker pointing to the last line that was executed.

Note: the debugger does not stop on a function call. The first line of the function is executed and the marker points to that line. When a **Fn.rtn** or **Fn.end** executes, the marker points to the function call.

For a description of the dialog box controls, see the **Debug.show** command, below.

Debug.show

Pauses the execution of the program and displays a dialog box. The dialog box will contain the result of the last **Debug.show.<command>** used or by default **Debug.show.program**.

There are three buttons in the dialog:

- Resume: Resumes execution.

- Step: Executes the next line while continuing to display the dialog box.

- View Swap: Opens a new dialog that allows you to choose a different Debug View.

The BACK key closes the debug dialog and stops your program.

Fonts

Your program can use fonts loaded from files.

At present, the only way to use a loaded font is with the **Gr.Text.SetFont** command.

Font.load <font_ptr_nvar>, <filename_sexp>

Loads a font from the file named by the <filename_sexp>. Returns a pointer to the font in the variable <font_ptr_nvar>. This pointer can be used to refer to the loaded font, for example, in a **Gr.Text.SetFont** command.

If the font file can not be loaded, the pointer will be set to -1. You can call the **GETERROR\$()** function to get an error message.

Font.delete {<font_ptr_nexp>}

Deletes the previously loaded font specified by the font pointer parameter <font_ptr_nexp>. Any variable that points to the deleted font becomes an invalid font pointer. An attempt to use the deleted font is an error. It is not an error to delete the same font again.

If the font pointer is omitted, the command deletes the most-recently loaded font that has not already been deleted. Repeating this operation deletes loaded fonts in last-to-first order. It is not an error to do this when there are no fonts loaded.

Font.clear

Clears the font list, deleting all previously loaded fonts. Any variable that points to a font becomes an invalid font pointer. An attempt to use any of the deleted fonts is an error.

Note: **Font.delete** leaves a marker in the font list, so pointers to other fonts will not be affected. That is why you can **Font.delete** the same font more than once. **Font.clear** clears the entire list, making all font pointer variables invalid. After executing **Font.clear**, you can't **Font.delete** any of the cleared fonts.

Console I/O

Output Console

BASIC! has three types of output screens: The Output Console, the Graphics Screen, and the HTML Screen. This section deals with the Output Console. See the section on Graphics for information about the Graphics Screen. See the section on HTML for information about the HTML screen.

Information is printed to screen using the Print command. BASIC! Run-time error messages are also displayed on this screen.

There is no random access to locations on this screen. Lines are printed one line after the other.

Although no line numbers are displayed, lines are numbered sequentially as they are printed, starting with 1. These line numbers refer to lines of text output, not to locations on the screen.

Print {<exp> {,;}} ...

? {<exp> {,;}} ...

Evaluates the expression(s) <exp> and prints the result(s) to the Output Console. You can use a question mark (?) in place of the command keyword **Print**.

If the comma (,) separator follows an expression then a comma and a space will be printed after the value of the expression.

If the semicolon (;) separator is used then nothing will separate the values of the expressions.

If the semicolon is at the end of the line, the output will not be printed until a **Print** command without a semicolon at the end is executed.

Print with no parameters prints a newline.

Examples:

PRINT "New", "Message"	% Prints: New, Message
PRINT "New"; " Message"	% Prints: New Message
PRINT "New" + " Message"	% Prints: New Message
? 100-1; " Luftballons"	% Prints: 99.0 Luftballons
? FORMAT\$("##", 99); " Luftballons"	% Prints: 99 Luftballons
PRINT "A"; "B"; "C";	% Prints: nothing
PRINT "D"; "E"; "F"	% Prints: ABCDEF

Print with User-Defined Functions

Note: Some commands, such as **Print**, can operate on either strings or numbers. Sometimes it has to try both ways before it knows what to do. First it will try to evaluate an expression as a number. If that fails, it will try to evaluate the same expression as a string.

If this happens, and the expression includes a function, the function will be called twice. If the function has side-effects, such as printing to the console, writing to a file, or changing a global parameter, the side-effect action will also happen twice.

Eventually this problem should be fixed in BASIC!, but until then you should be careful not to call a function, especially a user-defined function, as part of a **Print** command. Instead, assign the return value of the function to a variable, and then **Print** the variable. An assignment statement always knows what type of expression to evaluate, so it never evaluates twice.

```
! Do this:
y = MyFunction(x)
Print y
! NOT this:
Print MyFunction(x)
```

Cls

Clears the Output Console screen.

Console.front

Brings the Output Console to the front where the user can see it.

If BASIC! is running in the background with no screen visible, this command brings it to the foreground. If you have a different application running in the foreground, it will be pushed to the background.

If BASIC! is running in the foreground, but the Graphics or HTML screen is in the foreground, this command brings the Console to the foreground. BASIC! remains in Graphics or HTML mode.

Console.line.count <count_nvar>

Sets the return variable <count_nvar> to the number of lines written to the Console. This command waits for any pending Console writes to complete before reporting the count.

Console.line.text <line_nexp>, <text_svar>

The text of the specified line number of the Console is copied to the <text_svar>.

Console.line.touched <line_nvar> { <press_lvar> }

After an **OnConsoleTouch** interrupt indicates the user has touched the console, this command returns information about the touch.

The number of the line that the user touched is returned in the <line_nvar>.

If the optional <press_lvar> is present, the type of user touch—a short tap or a long press—is returned in the <press_lvar>. Its value will be 0 (false) if the touch was a short tap. Its value will be 1 (true) if the touch was a long press.

Console.save <filename_sexp>

The current contents of the Console is saved to the text file specified by the filename string expression.

Console.title { <title_sexp> }

Changes the title of the console window. If the <title_sexp> parameter is omitted, the title is changed to the default title, "BASIC! Program Output".

User Input and Interaction

This set of commands lets you interact with your programs.

At the lowest level, you can use **Inkey\$** to read raw keystrokes. You control display of the virtual keyboard with **Kb.hide**, **Kb.show**, and **Kb.toggle**.

With the **Select** command you present information in a list format that looks very much like the Output Console. If you prefer, you can use **Dialog.Select** to present the same information in a new dialog window. Either way, when your program runs, you select an item from the list by tapping a line.

The other commands in this group all pop up new windows.

Input lets you type a number or a line of text as input to your program. **Dialog.message** presents a message with a set of buttons to let you tell your program what to do next.

Popup is different. It presents information in a small, temporary display. It is not interactive and requires no management in your program. You pop it up and forget it.

The last two commands in this group present another kind of dialog window. The **Text.input** command operates on larger blocks of text, and **TGet** simulates terminal I/O.

Dialog.message {<title_sexp>}, {<message_sexp>}, <sel_nvar> {, <button1_sexp>{<button2_sexp>{, <button3_sexp>}}}

Generates a dialog box with a title, a message, and up to three buttons. When the user taps a button, the number of the selected button is returned in <sel_nvar>. If the user taps the screen outside of the message dialog or presses the BACK key, then the returned value is 0.

The string <title_sexp> becomes the title of the dialog box. The string <message_sexp> is displayed in the body of the dialog, above the buttons. The strings <button1_sexp>, <button2_sexp>, and <button3_sexp> provide the labels on the buttons.

You may have 0, 1, 2, or 3 buttons. On most devices, the buttons are numbered from right-to-left, because Android style guides recommend the positive action on the right and the negative action on the left. Some devices differ. On compliant devices, tapping the right-most button returns 1.

All of the parameters except the selection index variable <sel_nvar> are optional. If any parameter is omitted, the corresponding part of the message dialog is not displayed. Use commas to indicate omitted parameters (see Optional Parameters).

Examples:

```
Dialog.Message "Hey, you!", "Is this ok?", ok, "Sure thing!", "Don't care", "No way!"
Dialog.Message "Continue?", , go, "YES", "NO"
Dialog.Message , "Continue?", go, "YES", "NO"
Dialog.Message , , b
```

The first command displays a full dialog with a title, a message, and three buttons.

The second command displays a box with a title and two buttons – note that the **YES** button will be on the right and the **NO** button on the left. The third displays the same information, but it looks a little different because the text is displayed as the message and not as the title. Note the commas.

The fourth command displays nothing at all. The screen dims and your program waits for a tap or the BACK key with no feedback to tell the user what to do.

Dialog.select <sel_nvar>, <Array\$[]>|<list_nexp> {<title_sexp>}

Generates a dialog box with a list of choices for the user. When the user taps a list item, the index of the selected line is returned in the <sel_nvar>. If the user taps the screen outside of the selection dialog or presses the BACK key, then the returned value is 0.

<Array\$[]> is a string array that holds the list of items to be selected. The array is specified without an index but must have been previously dimensioned or loaded via Array.load.

As an alternative to an array, a string-type list may be specified in the <list_nexp>.

The <title_sexp> is an optional string expression that will be displayed at the top of the selection dialog. If the parameter is not present, or the expression evaluates to an empty string (""), the dialog box will be displayed with no title.

This command also accepts optional <message_sexp> and <press_lvar> parameters like those described in the **Select** command, but they should not be used. The <message_sexp> is ignored and the <press_lvar> will always be set to 0.

Input {<prompt_sexp>}, <result_var>{ {<default_exp>}{, <canceled_nvar>}}

Generates a dialog box with an input area and an **OK** button. When the user taps the button, the value in the input area is written to the variable <result_var>.

The <prompt_sexp> will become the dialog box title. If the prompt expression is empty ("") or omitted, the dialog box will be drawn without a title area.

If the return variable <result_var> is numeric, the input must be numeric, so the only key taps that will be accepted are 0-9, "+", "-", and ".". If <result_var> is a string variable, the input may be any string.

If a <default_exp> is given then its value will be placed into the input area of the dialog box. The default expression type must match the <result_var> type.

The variable <canceled_nvar> controls what happens if the user cancels the dialog, either by tapping the BACK key or by touching anywhere outside of the dialog box.

If you provide a <canceled_nvar>, its value is set to **false** (0) if the user taps the **OK** button, and **true** (1) if the users cancels the dialog.

If you do not provide a <canceled_nvar>, a canceled dialog is reported as an error. Unless there is an "OnError:" the user will see the messages:

```
Input dialog cancelled
Execution halted
```

If there is an "OnError:" label, execution will resume at the statement following the label.

The <result_var> parameter is required. All others are optional. These are all valid:

```
INPUT "prompt", result$, "default", isCanceled
INPUT , result$, "default"
INPUT "prompt", result$, , isCanceled
INPUT "prompt", result$
INPUT , result$
```

Note the use of commas as parameter placeholders (see Optional Parameters).

Inkey\$ <svar>

Reports key taps for the a-z, 0-9, Space and the D-Pad keys. The key value is returned in <svar>.

The D-Pad keys are reported as "up", "down", "left", "right" and "go". If any key other than those have been tapped, the string "key nn" will be returned. Where nn will be the Android key code for that key.

If no key has been tapped, the "@" character is returned in <svar>.

Rapid key taps are buffered in case they come faster than the BASIC! program can handle them.

Popup <message_sexp> {{ <x_nexp> }} <y_nexp> {{ <duration_lexp> }}

Pops up a small message for a limited duration. The message is <message_sexp>.

All of the parameters except the message are optional. If omitted, their default values are 0. Use commas to indicate omitted parameters (see Optional Parameters).

The simplest form of the **Popup** command, **Popup "Hello!"**, displays the message in the center of the screen for two seconds.

The default location for the Popup is the center of the screen. The optional <x_nexp> and <y_nexp> parameters give a displacement from the center. The values may be negative.

Select the duration of the Popup, either 2 seconds or 4 seconds, with the optional <duration_lexp> "long flag". If the flag is false (the expression evaluates to 0) the message is visible for 2 seconds. If the flag is true (non-zero) the message is visible for 4 seconds. If the flag is omitted the duration is short.

Select <sel_nvar>, <Array\$[]>|<list_nexp> {,<title_sexp> {, <message_sexp> } } {<press_lvar> }

The Select command generates a new screen with a list of choices for the user. When the user taps a screen line, the index of the selected line is returned in the <sel_nvar>. If the user presses the BACK key, then the returned value is 0.

<Array\$[]> is a string array that holds the list of items to be selected. The array is specified without an index but must have been previously dimensioned, loaded via Array.load, or created by another command.

As an alternative to an array, a string-type list may be specified in the <list_nexp>.

The <title_sexp> is an optional string expression that is placed into the title bar at the top of the selection screen. If the parameter is not present, the screen displays a default title. If the expression evaluates to an empty string ("") the title is blank.

The <message_sexp> is an optional string expression that is displayed in a short Popup message. If the message is an empty string ("") there is no Popup. If the parameter is absent, the <title_sexp> string is used instead, but if the <title_sexp> is also missing or empty, there is no Popup.

The <press_lvar> is optional. If present, the type of user tap—long or short—is returned in <press_lvar>. The value returned is 0 (false) if the user selected the item with a short tap. The value returned is 1 (true) if the user selected the item with a long press.

Use commas to indicate omitted optional parameters (see Optional Parameters).

Text.input <svar>{ { <text_sexp> } , <title_sexp> }

This command is similar to "Input" except that it is used to input and/or edit a large quantity of text. It opens a new window with scroll bars and full text editing capabilities. You may set the title of the new window with the optional <title_sexp> parameter.

If the optional <text_sexp> is present then that text is loaded into the text input window for editing. If <text_sexp> is not present then the text.input text area will be empty. If <title_sexp> is needed but text.input text area is to be initially empty, use two commas to indicate the <sexp> specifies the title and not the initial text.

When done editing, tap the Finish button. The edited text is returned in <svar>.

If you tap the BACK key then all text editing is discarded. <svar> returns the original <sexp> text.

The following example grabs the Sample Program file, **f01_commands.bas**, to string s\$. It then sends s\$ to text.input for editing. The result of the edit is returned in string r\$. r\$ is then printed to console.

```
GRABFILE s$, "../source/ Sample_Programs/f01_commands.bas"
TEXT.INPUT r$,s$
PRINT r$
END
```

TGet <result_svar>, <prompt_sexp> {, <title_sexp>}

Simulates a terminal. The current contents of the Output Console is displayed in a new window. The last line displayed starts with the prompt string followed by the cursor. The user types in the input and taps enter. The characters that the user typed in is returned in <result_svar>. The prompt and response are displayed on the Output Console.

You may set the title of the text input window with the optional <title_sexp> parameter.

Kb.hide

Hides the soft keyboard.

If the keyboard is showing, and you have an **OnKbChange:** interrupt label, BASIC! will jump to your interrupt label when the keyboard closes.

The soft keyboard is always hidden when your program starts running, regardless of whether it is showing in the Editor.

Note: BASIC! automatically hides the soft keyboard when you change screens. For example, if the keyboard is showing over the Output Console, and you execute **Gr.open** to start Graphics Mode, the keyboard is hidden. The keyboard will not be showing when you exit Graphics Mode and return to the Console. Similarly, if you show the keyboard over your Graphics screen and you execute **Gr.close** to return to the Console, the keyboard is hidden.

If you have an **OnKbChange:** interrupt label, automatically hiding the keyboard does **not** trigger a jump to the interrupt label.

Kb.show

Shows the soft keyboard.

If the keyboard is not showing, and you have an **OnKbChange:** interrupt label, BASIC! will jump to your interrupt label when the keyboard opens.

When the soft keyboard is showing, its keys may be read using the **Inkey\$** command. The command may not work in devices with hard or slide-out keyboards.

You cannot show the soft keyboard over the Output Console unless you first **Print** to the Console.

Kb.toggle

Toggles the showing or hiding of the soft keyboard. If the keyboard is being shown, it will be hidden. If it is hidden, it will be shown.

Kb.showing <lvar>

Reports the visibility of the soft keyboard. If the keyboard is showing, the <lvar> is set to 1.0 (true), otherwise the <lvar> is set to 0.0 (false).

This command reports only the status of the keyboard shown by **Kb.show**. For example, the keyboard attached to the **Input** command dialog box cannot be controlled by **Kb.show** or **Kb.hide** and its status is not reported by **Kb.showing**.

OnKbChange:

If you show a soft keyboard with **Kb.show**, or close that same keyboard with **Kb.hide** or by tapping the BACK key, the change takes some time. The keyboard may open or close a few hundred milliseconds after it is requested. **Kb.show** and **Kb.hide** block until the change is complete, but your program does not know when you tap the BACK key.

If you have an **OnKbChange:** interrupt label in your program, then when the keyboard changes as just described, BASIC! interrupts your program and executes the statements after the interrupt label.

To return control to where the interrupt occurred, execute **Kb.resume** in your interrupt handler.

This interrupt occurs only for keyboards you show with **Kb.show**. Keyboards attached to other screens, such as **TGet** or the **Input** dialog box, do not cause **OnKbChange:** interrupts.

Kb.resume

Resume program execution at the point where the **OnKbChange:** interrupt occurred.

When an interrupt occurs, no other interrupt can occur (except **OnError:**) until the corresponding **resume** executes.

The Soft Keyboard and the BACK Key

If you show a soft keyboard with **Kb.show**, and you tap the BACK key, the keyboard closes. The current screen (Console or Graphics screen) does not close. BASIC! does not trap the keypress with either **OnBackKey:** or **OnKeyPress:**. You can use the **OnKbChange:** interrupt label to be notified that the keyboard closed.

Working with Files

Android devices may have several file storage devices. BASIC! uses one of these devices as its *base drive*. You can select a different base drive in the *Menu->Preferences* item *BASE DRIVE*. In this manual, the notation **<pref base drive>** refers to the base drive you selected in Preferences.

BASIC! can work with files anywhere on the base drive, but most file operations are done in BASIC!'s *base directory*. Except when you create a standalone apk file (see Appendix D), the base directory is **<pref base drive>/rfo-basic**. All file paths are relative to a subdirectory of the base directory.

Paths Explained – a beginner's guide

A path describes where a file or directory is located relative to another directory.

A file is a container for data. A directory is a container for files and other directories.¹ A directory that is in a directory may have other directories in it, and those directories may contain other directories, and so on. This results in a "tree" of directories and files, called a *file system*. A file system organizes data on a storage device, such as a disk or a memory card.

Absolute paths: A path that is relative to the root directory is called an *absolute path*. For example, the absolute path to pictures you take with an Android camera may be `/sdcard/DCIM/Camera`. In BASIC!, the *base directory* is not a *root directory*, so *BASIC! does not use absolute paths*.

Relative paths: A path that is relative to anything except the root directory is called a *relative path*. Starting from `/sdcard`, the relative path down to `Camera` is `DCIM/Camera`. Use the `../` path notation to go back up: from `Camera`, the path to `/sdcard/DCIM` is `../` and to `/sdcard` is `../../`.

The relative path from `/sdcard/DCIM/Camera` to `/sdcard/DCIM/.thumbnails` is `../.thumbnails`. With this notation, you can reach any file in the file system.

All paths in BASIC! are relative to a default path that depends on the kind of data you want to use. These default paths are explained in the next section.

Paths in BASIC!

BASIC! files are stored in subdirectories of the base directory, **<pref base drive>/rfo-basic/**. Files are grouped by type, as follows:

- BASIC! program files are in **rfo-basic/source/**
- BASIC! data files are in **rfo-basic/data/**
- BASIC! SQLite databases are in **rfo-basic/databases/**

All of the BASIC! file commands assume a certain default path. The default path depends on the type of file each command expects to handle:

¹ This container is called a "directory" because it is a listing of the items it contains. You can also call it a "folder", and you can say "a folder is a container for files and other folders." Both expressions mean the same thing.

- The **Include** and **Run** commands expect to load program files, so they look in **rfo-basic/source/**.
- **SQLite** operations look for database files in **rfo-basic/databases/**.
- All other file operations look for data files in **rfo-basic/data/**.

If you give a filename to a file command, it looks for that filename in the default directory for commands of that type. If you want to work with a file that is not in that directory, you must specify a relative path to the appropriate directory. BASIC! adds your path to the default path.

- To read the file "names.txt" in "rfo-basic/data/", the path is "names.txt".
- To read the program file "sines.bas" in "rfo-basic/source", the path is "../source/sines.bas".

Paths Outside of BASIC!

BASIC! runs on an Android device, and its files are part of the Android file system.

You can use relative paths to access files outside of the base directory. To continue the example from the previous section, assume the absolute Android path to the <pref base drive> is "/sdcard":

- To read the music file "rain.mp3" in "/sdcard/music/", the BASIC! path is "../music/rain.mp3". The path is relative to the default directory for general data "<pref base drive>/rfo-basic/data/".

Use care when writing paths that look up from the <pref base drive>. The directory name may not be what you expect, and it may not be the same on all Android devices.

Every file on an Android device has an absolute path. Unfortunately, on most Android devices every file has several absolute paths. The default <pref base drive> can be reached with the absolute Android path "/sdcard". The name "sdcard" is a shortcut (a *symbolic link*) that means different things on different devices. "<pref base drive>/rfo-basic/.." may not get to the Android directory "/sdcard", but to something like "/storage/emulated/legacy".

You can use the command **File.root** to get the absolute Android path to the BASIC! <pref base drive>.

Paths and Case-sensitivity

While the Android file system is normally case-sensitive. However, the FAT file system, often used on SD cards, memory sticks, etc., is case-insensitive. When handling files on these devices, Android – and therefore BASIC! – can not differentiate between names that differ only in case. In your BASIC! program, the two paths "../music/rain.mp3" and "../MUSIC/Rain.MP3" will both access the same file.

The rules change if you compile the same BASIC! program into a standalone apk (see Appendix D). The file system inside the apk is case-sensitive. The paths "../music/rain.mp3" and "../MUSIC/Rain.MP3" access different files. If the actual path in your build project is "Assets/<project>/MUSIC/Rain.MP3", then using the second path would succeed, but the first path would fail.

To prevent any error, it is good practice to match case exactly in file paths and names.

Mark and Mark Limit

Note: This is an advanced file management technique that you will rarely need to use. However, see the note below about Out of Memory errors when reading very large files.

Every file has a mark and a mark limit. The mark is a position, and the mark limit is a size. As you read a file, its data is copied into a buffer. The buffer starts at the mark, and its length is the mark limit. BASIC! uses the buffer to allow you to reposition within the file. You are really repositioning within the buffer.

If you do not mark a file, then the first time you read it or set a position in it, the mark is set at position 1 and the mark limit is set to the size of the file. This allows you to position and read anywhere in the file.

The **Text.position.mark** and **Byte.position.mark** commands override the default mark and mark limit. You can change the mark and mark limit as often as you like, but there is only one mark in a file.

You cannot set a position before the mark. If you try, the file will be positioned at the mark. You will not be notified that the current position is different from what you requested, but you can use **Text/Byte.position.get** to determine the real position. Since the default mark position is 1, you can position anywhere if you never set a mark.

You cannot make the buffer smaller. If you want the buffer to be smaller than the file, you must execute **Text/Byte.position.mark** before reading the file or setting a position. The smallest buffer size available is 8096 bytes, but it is not an error to specify a smaller number. Since the default mark limit is the file size, you can position anywhere if you never set a mark limit.

If you read or position past the end of the buffer (more than **marklimit** bytes beyond the the mark), the mark is invalid. It is an error to try to move the position backward when the mark is invalid. You will see the error message "**Invalid mark**". Since the default buffer is the whole file, you will never see this error if you never set a mark.

There is only one condition that requires you to use this command. **If you open a very large file, the default buffer size may be too large.** Reading or positioning to the end of the file may cause an **Out Of Memory** error. To avoid this error, your must use **Text.position.mark** to reduce the buffer size.

File Commands

File.delete <lvar>, <path_sexp>

The file or directory at <path_sexp> will be deleted, if it exists. If the file or directory did not exist before the Delete, the <lvar> will contain zero. If the file or directory did exist and was deleted, the <lvar> will be returned as not zero.

The default path is "<pref base drive>/rfo-basic/data/".

File.dir <path_sexp>, Array\$[] {<dirmark_sexp>}

Returns the names of the files and directories in the path specified by <path_sexp>. The path is relative to "<pref base drive>/rfo-basic/data/".

The names are placed into Array\$[]. The array is sorted alphabetically with the directories at the top of the list. If the array exists, it is overwritten, otherwise a new array is created. The result is always a one-dimensional array.

A directory is identified by a marker appended to its name. The default marker is the string "(d)". You can change the marker with the optional directory mark parameter <dirmark_sexp>. If you do not want directories to be marked, set <dirmark_sexp> to an empty string, "".

Dir is a valid alias for this command.

File.exists <lvar>, <path_sexp>

Reports if the <path_sexp> directory or file exists. If the directory or file does not exist, the <lvar> will contain zero. If the file or directory does exist, the <lvar> will be returned as not zero.

The default path is "<pref base drive>/rfo-basic/data/".

File.mkdir <path_sexp>

Before you can use a directory, the directory must exist. Use this command to create a directory named by the path string <path_sexp>.

The new directory is created relative to the default directory "<pref base drive>/rfo-basic/data/". For example:

- To create a new directory, "homes", in "<pref base drive>/rfo_basic/data/", use the path "homes/", or simply "homes".
- To create a new directory, "icons", in the root directory of the SD card, use ".././icons".

Mkdir is a valid alias for this command.

File.rename <old_path_sexp>, <new_path_sexp>

The file or directory at old_path is renamed to new_path. If there is already a file present named <new_path_sexp>, it is silently replaced.

The default path is "<pref base drive>/rfo-basic/data/".

The rename operation can not only change the name of a file or a directory, it can also move the file or directory to another directory.

For example:

```
Rename ".././testfile.txt", "testfile1.txt"
```

removes the file, testfile.txt, from "<pref base drive>/", places it into "sdcard/rfo-basic/data/" and also renames it to testfile1.txt.

Rename is a valid alias for this command.

File.root <svar>

Returns the canonical path from the file system root to "<pref base drive>/rfo-basic/data", the default data directory, in <svar>. The <pref base drive> is expanded to the full absolute Android path from the file system root, "/".

You cannot use this path in any BASIC! command, as BASIC! paths are relative to a command-dependent default directory. However, you can use it to compute a relative path to parts of the Android file system outside of the BASIC! <pref base drive>.

File.size <size_nvar>, <path_sexp>

The size, in bytes, of the file at <path_sexp> is returned in <size_nvar>. If there is no file at <path_sexp>, this command generates a run-time error.

The <path_sexp> is appended to the default path, "<pref base drive>/rfo-basic/data/".

File.type <type_svar>, <path_sexp>

Returns a one-character type indicator in <type_svar> for the file at <path_sexp>. The <path_sexp> is appended to the default data path, "<pref base drive>/rfo-basic/data/". The type indicator values are:

Indicator:	Meaning:
"d"	"directory" – path names a directory
"f"	"file" – path names a regular file
"o"	"other" – path names a special file
"x"	file does not exist

Text File I/O

The text file I/O commands are to be exclusively used for text (.txt) files. Text files are made up of lines of characters that end in CR (Carriage Return) and/or NL (New Line). The text file input and output commands read and write entire lines.

The default path is "<pref base drive>/rfo-basic/data/".

Text.open {r|w|a}, <file_table_nvar>, <path_sexp>

The file specified by the path string expression <path_sexp> is opened. The default path is "<pref base drive>/rfo-basic/data/". The <path_sexp> string is appended to the default path.

The first parameter is a single character that sets the I/O mode for this file:

Parameter	Mode	Notes
r	read	File exists: Reads from the start of the file. File does not exist: Error (see below).
w	write	File exists: Writes from the start of the file. Writes over any existing data. File does not exist: Creates a new file. Writes from the start of the file.
a	append	File exists: Writing starts after the last line in the file. File does not exist: Creates a new file. Writes from the start of the file.

A file table number is placed into the numeric variable <file_table_nvar>. This value is for use in subsequent **Text.readln**, **Text.writeln**, **Text.eof**, **Text.position.***, or **Text.close** commands.

If a file being opened for read does not exist then the <file_table_nvar> will be set to -1. The BASIC! programmer can check for this and either create the file or report the error to the user. Information about the error is available from the **GETERROR\$()** function.

Opening a file for append that does not exist creates an empty file. Finally, opening a file for write that already exists deletes the contents of the file; that is, it replaces the existing file with a new, empty one.

Text.close <file_table_nexp>

The previously opened file represented by <file_table_nexp> will be closed.

Note: It is essential to close an output file if you have written over 8k bytes to it. If you do not close the file then the file will only contain the first 8k bytes.

Text.readln <file_table_nexp> {<svar>}...

Read the lines from the specified, previously opened file and write them into the <svar> parameter(s). If <file_table_nexp> is -1, indicating **Text.open** failed, or if it is not a valid file table number, then the command throws a run-time error.

After the last line in the file has been read, further attempts to read from the file place the characters "EOF" into the <line_svar> parameter(s). This is indistinguishable from the string "EOF" read as actual data, except that the result of the **Text.eof** command will be false after reading real data and true after reading at end-of-file (EOF).

This example reads an entire file and prints each line.

```
TEXT.OPEN r, file_number, "testfile.txt"
DO
  TEXT.READLN file_number, line$
  PRINT line$
UNTIL line$ = "EOF"
TEXT.CLOSE file_number
```

The file will not automatically be closed when the end-of-file is read. Subsequent reads from the file will continue to return "EOF".

When you are done reading a file, the **Text.close** command should be used to close the file.

Text.writeln <file_table_nexp>, <parms same as Print>

The parameters that follow the file table pointer are parsed and processed exactly the same as the **Print** command parameters. This command is essentially a **Print** to a file.

If a parameter line ends with a semicolon, the data is not written to the file. It is stored in a temporary buffer until the next **Text.writeln** command that does not end in a semicolon. There is only one temporary buffer no matter how many files you have open. If you want to build partial print lines for more than one file at a time, do not use **Text.writeln** commands ending with semicolons. Instead use string variables to store the temporary results.

After the last line has been written to the file, the **Text.close** command should be used to close the file.

Text.writeln with no parameters writes a newline.

Text.eof <file_table_nexp>, <lvar>

Report an opened file's end-of-file status. If the file is at EOF, the <lvar> is set true (non-zero). If the file or directory is not at EOF, the <lvar> is set false (zero).

A file opened for write or append is always at EOF. A file opened for read is not at EOF until all of the data has been read and then one more read is attempted. That read will have returned the string "EOF". **Text.position.set** may also position the file at EOF.

Text.position.get <file_table_nexp>, <position_nvar>

Get the position of the next line to be read or written to the file. The position of the first line in the file is 1. The position is incremented by one for each line read or written. The position information can be used for setting up random file data access.

Note: If a file is opened for append, the position returned will be relative to the end of the file. The position returned for the first line to be written after a file is opened for append will be 1. You will have to add these new positions to the known position of the end of the file when building your random access table.

Text.position.set <file_table_nexp>, <position_nexp>

Sets the position of the next line to read. A position value of 1 will read the first line in the file.

Text.position.set can only be used for files open for text reading.

If the position value is greater than the number of lines in the file, the file will be positioned at the end of file. The position returned for **Text.position.get** at the EOF will be number of lines plus one in the file.

If you have marked a position in the file, you cannot set a position before the mark. You will not be notified that the position is different from what you requested. See **Text.position.mark** for more information.

Text.position.mark {{<file_table_nexp>},{ <marklimit_nexp>}}

Marks the current line of the file, and sets the mark limit to <marklimit_nexp> bytes.

Both parameters are optional. If the file table index <file_table_nexp> is omitted, the default file is the last file opened; you must ensure that the last file opened was a **Text** file opened for reading. If the mark limit <marklimit_exp> is omitted, the default value is the file's current mark limit.

Please read **Working with Files → Mark and Mark Limit**, above.

GrabURL <result_svar>, <url_sexp>{, <timeout_nexp>}

Copies the entire source text of the URL <url_sexp> to the string variable <result_svar>. The URL may specify an Internet resource or a local file. If the URL does not exist or the data cannot be read, the <result_svar> is set to the empty string, "", and you can use the **GETERROR\$()** function to get more information.

If the optional <timeout_nexp> parameter is non-zero, it specifies a timeout in milliseconds. This is meaningful only if the URL names a resource on a remote host. If the timeout time elapses and host does not connect or does not return any data, **GETERROR\$** reports a socket timeout.

If the named resource is empty, the <result_svar> is empty, "", and **GETERROR\$()** returns "No error".

The **Split** command can be used to split the <result_svar> into an array of lines.

GrabFile <result_svar>, <path_sexp>{, <unicode_flag_lexp>}

Copies the entire contents of the file at <path_sexp> to the string variable <result_svar>. By default, **GrabFile** assumes that the file contains binary bytes or ASCII characters. If the optional <unicode_flag_lexp> evaluates to true (a non-zero numeric value), **GrabFile** can read Unicode text.

If the file does not exist or cannot be opened, the <result_svar> is set to the empty string, "", and you can use the **GETERROR\$()** function to get more information. If the file is empty, the <result_svar> is an empty string, "", but **GETERROR\$()** returns "No error".

For text files, either ASCII or Unicode, the **Split** command can be used to split the <result_svar> into an array of lines. **GrabFile** can also be used grab the contents of a text file for direct use with **Text.input**:

```
GRABFILE text$, "MyJournal.txt"  
TEXT.INPUT EditedText$, text$
```

Byte File I/O

Byte file I/O can be used to read and write any type of file (.txt, .jpg, .pdf, .mp3, etc.). Each command reads or writes one byte or a sequence of bytes as binary data.

Byte.open {r|w|a}, <file_table_nvar>, <path_sexp>

The file specified by the path string expression <path_sexp> is opened. If the path is a URL starting with "http..." then an Internet file is opened. Otherwise, the <path_sexp> string is appended to the default path "<pref base drive>/rfo-basic/data/".

The first parameter is a single character that sets the I/O mode for this file:

Parameter	Mode	Notes
r	read	File exists: Reads from the start of the file. File does not exist: Error (see below).
w	write	File exists: Writes from the start of the file. Writes over any existing data. File does not exist: Creates a new file. Writes from the start of the file.
a	append	File exists: Writing starts after the last line in the file. File does not exist: Creates a new file. Writes from the start of the file.

A file table number is placed into the numeric variable <file_table_nvar>. This value is for use in subsequent **Byte.read.***, **Byte.write.***, **Byte.eof**, **Byte.position.***, **Byte.truncate**, **Byte.copy**, or **Byte.close** commands.

If a file being opened for read does not exist then the <file_table_nvar> will be set to -1. The BASIC! program can check for this and either create the file or report the error to the user. Information about the error is available from the **GETERROR\$()** function.

Byte.close <file_table_nexp>

Closes the previously opened file.

Byte.read.byte <file_table_nexp> {<nvar>}...

Reads bytes from a file. The <file_table_nexp> parameter is a file table number returned by a previous **Byte.open**. If the file number is -1 then the command throws a run-time error.

Places the byte(s) into the <nvar> parameter(s) as positive values, $0 \leq \text{byte} \leq 255$. After the last byte in the file has been read, further attempts to read from the file return the value -1. The result of the **Byte.eof** command will be false after reading real data and true after reading at end-of-file (EOF).

This example reads a file and prints each byte, and prints "-1" at the end to show that the entire file has been read.

```
BYTE.OPEN r, file_number, "testfile.jpg"
DO
  BYTE.READ.BYTE file_number, Byte
```



```
PRINT Byte
UNTIL Byte < 0
BYTE.CLOSE file_number
```

Byte.write.byte <file_table_nexp> {{,<nexp>}...{<sexp>}}

Writes bytes to the file specified by the file number parameter <file_table_nexp>. The bytes are written from an optional comma-separated list of numeric expressions, a single optional string expression, or both.

- Numeric parameters: the command writes the low-order 8 bits of the value of each expression as a single byte.
- String parameters: the command writes the entire string to the file. Each character of the string is written as a single byte. See **Byte.write.buffer** for more information.
- If you have both numeric and string parameters, you may have only one string, and it must be last.
- The command accepts a string for backward compatibility. **Byte.write.buffer** is preferred.

Examples:

```
Byte.open w, f1, "tmp.dat"      % create a file
Byte.write.byte f1, 10          % write one byte
Byte.write.byte f1, 11, 12, 13  % write three bytes
Byte.write.byte f1, "Hello!"    % write six bytes
Byte.write.byte f1, 1,2,3,"abc" % write six bytes
Byte.write.byte f1, "one", "two" % syntax error: only one string allowed
```

Note: If the bytes are written from a string expression then the expression is evaluated twice. You should not put calls to user-defined functions in the expression.

Byte.read.number <file_table_nexp> {,<nvar>}...

Reads numbers from the file specified by the file number parameter <file_table_nexp> and places them into the numeric variables in the "{,<nvar>}..." parameter list. Each number is a group of 8 bytes.

If the file does not have enough data available for all of the variables, the value of one or more <nvar> will be set to -1. This is indistinguishable from -1 read as actual data, except that the result of the **Byte.eof** command will be false for real data and true for EOF.

Normally this command is used only to read values written with **Byte.write.number**. You must be sure the file is positioned at the first byte of the eight-byte representation of a number, or you will get unexpected results.

Byte.write.number <file_table_nexp> {,<nexp>}...

Writes the values of the numeric expressions <nexp> to the file specified by the file number parameter <file_table_nexp>. This command always writes 8 bytes for each expression in the parameter list.

Byte.read.buffer <file_table_nexp>, <count_nexp>, <buffer_svar>

Reads the specified number of bytes (<count_nexp>) into the buffer string variable (<buffer_svar>) from the file. The string length (len(<buffer_svar>)) will be the number of bytes actually read. If the end of file is reached, the string length may be less than the requested count.

A buffer string is a special use of the BASIC! string. Each character of a string is 16 bits. When used as a buffer, one byte of data is written into the lower 8 bits of each 16-bit character. The upper 8 bits are 0. Extract the binary data from the string, one byte at a time, with the **ASCII()** or **UCODE()** functions.

The format of the buffer string read by this command is compatible with the **DECODE\$()** function. If you know that part of your data contains an encoded string, you can extract the substring (using a function like **MID\$()**), then pass the substring to **DECODE\$()** to convert it to a BASIC! string.

Byte.write.buffer <file_table_nexp>, <buffer_sexp>

Writes the entire contents of the string expression to the file. The string is assumed to be a buffer string holding binary data, as described in **Byte.read.buffer**. The writer discards the upper 8 bits of each 16-bit character, writing one byte to the file for each character in the string.

The **Byte.read.buffer** command and the **ENCODE\$()** function always create these "buffer strings". You can construct one by using, for example, the **CHR\$()** function with values less than 256.

If you use only ASCII characters in a string, you can use this function to write the string to a file. The output is the same as if you had written it with **Text.writeIn**, except that it will have no added newline.

Byte.eof <file_table_nexp>, <lvar>

Reports an opened file's end-of-file status. If the file is at EOF, the <lvar> is set true (non-zero). If the file or directory is not at EOF, the <lvar> is set false (zero).

A file opened for write or append is always at EOF. A file opened for read is not at EOF until all of the data has been read and then one more read is attempted. That read will have returned the value -1. **Byte.position.set** may also position the file at EOF.

Byte.position.get <file_table_nexp>, <position_nvar>

Gets the position of the next byte to be read or written. The position of the first byte is 1. The position value will be incremented by 1 for each byte read or written.

The position information can be used for setting up random file data access.

If the file is opened for append, the position returned will be the length of the file plus one.

Byte.position.set <file_table_nexp>, <position_nexp>

Sets the position of the next byte to be read from the file. If the position value is greater than the position of the last byte of the file, the position will point to the end of file.

This command can only be used on files open for byte read.

Byte.position.mark {{<file_table_nexp>},{ <marklimit_nexp>}}

Marks the current position in the file, and sets the mark limit to <marklimit_nexp> bytes.

Both parameters are optional. If the file table index <file_table_nexp> is omitted, the default file is the last file opened; you must ensure that the last file opened was a **Byte** file opened for reading. If the mark limit <marklimit_exp> is omitted, the default value is the file's current mark limit.

Please read **Working with Files → Mark and Mark Limit**, above.

Byte.truncate <file_table_nexp>,<length_nexp>

Truncates the file to <length_nexp> bytes and closes the file. Setting the truncate length <length_nexp> larger than the current length (current write position - 1) has no effect.

This command can only be used on files open for byte write or append.

Byte.copy <file_table_nexp>,<output_file_sexp>

Copies the previously open input file represented by <file_table_nexp> to the file whose path is specified by <output_file_sexp>. The default path is "<pref base drive>/rfo-basic/data/".

If <file_table_nexp> = -1 then a run-time error will be thrown.

All bytes from the current position of the input file to its end are copied to the to the output file. Both files are then closed.

If you have read from the input file, and you want to copy the whole file, you must reset the file position to 0 with **Byte.position.set**. However, if you have changed the file mark with **Byte.position.mark**, or if you reading a non-local (internet) file, you can't reset the file position to 0. Instead, you must close and reopen the file.

You should use **Byte.copy** if you are using Byte I/O for the sole purpose of copying. It is thousands (literally) of times faster than using **Byte.read/Byte.write**.

ZIP File I/O

The ZIP file I/O commands work with compressed files. ZIP is an archive file format that stores multiple directories and files, using a method of lossless data compression to save file space.

Use **Zip.dir** to get an array containing the names of all of the directories and files in an archive. Use the file names with **Zip.read** to extract files from the archive. **Zip.read** can not extract a directory. Use **Zip.write** to put files in a new archive. You can overwrite an existing ZIP file, but you cannot replace or add entries.

Zip.count <path_sexp>, <nvar>

Returns the number of entries inside the ZIP file located at <path_sexp>. The path is relative to "<pref base drive>/rfo-basic/data/".

The count is returned in the <nvar>. If the ZIP file does not exist, the returned count is 0.

Zip.dir <path_sexp>, Array\$[] {,<dirmark_sexp>}

Returns the names of the files and directories inside the ZIP file located at <path_sexp>. The path is relative to "<pref base drive>/rfo-basic/data/".

The names are placed into Array\$[]. The array is sorted alphabetically with the directories at the top of the list. If the array exists, it is overwritten, otherwise a new array is created. The result is always a one-dimensional array.

A directory is identified by a marker appended to its name. The default marker is the string "(d)". You can change the marker with the optional directory mark parameter <dirmark_sexp>. If you do not want directories to be marked, set <dirmark_sexp> to an empty string, "".

Zip.open {r|w|a}, <file_table_nvar>, <path_sexp>

The ZIP file specified by the path string expression <path_sexp> is opened. The path is relative to "<pref base drive>/rfo-basic/data/".

The first parameter is a single character that sets the I/O mode for this file:

Parameter	Mode	Notes
r	read	File exists: Reads from the start of the file. File does not exist: Error (see below).
w	write	File exists: Writes from the start of the file. Writes over any existing data. File does not exist: Creates a new file. Writes from the start of the file.

Note: unlike **Text.open** and **Byte.open**, **Zip.open** does not support append mode.

A file table number is placed into the numeric variable <file_table_nvar>. This value is for use in subsequent **Zip.read**, **Zip.write**, or **Zip.close** commands.

If there was an error opening the ZIP file, <file_table_nvar> is set to -1 with details available from the **GETERROR\$()** function.

Zip.close <file_table_nexp>

Closes the previously opened ZIP file.

Zip.read <file_table_nexp>, <buffer_svar>, <file_name_sexp>

Reads the content of the file <file_name_sexp> from inside a ZIP file and puts the result byte(s) inside the buffer string variable <buffer_svar>.

The <file_table_nexp> parameter is a file table number returned by a previous **Zip.open** command. If the file number is -1 then the command throws a run-time error.

If the file <file_name_sexp> is not found in the ZIP, <buffer_svar> is set to "EOF".

If the user tries to read the content of a zipped directory, instead of a zipped file, then the command throws a run-time error. To read the contents of a zipped directory, use **Zip.dir**.

Zip.write <file_table_nexp> ,<buffer_sexp> ,<file_name_sexp>

Writes the entire contents of the string expression <buffer_sexp> into a ZIP, as a file named <file_name_sexp>. The ZIP is in a file previously opened with **Zip.open**.

The string <buffer_sexp> is assumed to be a buffer string holding binary data, typically a string coming from reading a local file with **Byte.read.buffer**.

HTML

Introduction

The BASIC! HTML package is designed to allow the BASIC! programmer to create user interfaces using HTML and JavaScript. The interface provides for interaction between the HTML engine and BASIC!. The HTML programmer can use JavaScript to send messages to BASIC!. The HTML engine will also report user events such as the BACK key, hyperlink transfers, downloads, form data and errors.

The demo program, **f37_html_demo.bas**, combined with the HTML demo files, **htmlDemo1.html** and **htmlDemo2.html**, illustrate the various commands and possibilities. The content of all three files are listed in the **Appendix B** of this document. They are also delivered with the BASIC! apk. It is highly recommended that all three files be carefully studied to fully understand this interface.

Another demo program, **f38_html_edit.bas**, can be used to edit html files. To use the program, run it and enter the html file name without the html extension. The program will add the extension ".html".

Caution: Forum users have reported problems in HTML mode with commands that use a new window or screen to interact with the user (**Input**, **Select** and others), or when using the BACK key to try to control actions outside of the HTML WebView.

HTML Commands

Html.open {<ShowStatusBar_lexp> {, <Orientation_nexp>}}

This command must be executed before using the HTML interface.

The Status Bar will be shown on the Web Screen if the <ShowStatusBar_lexp> is true (not zero). If the <ShowStatusBar_lexp> is not present, the Status Bar will not be shown.

The orientation upon opening the HTML screen will be determined by the <Orientation_nexp> value. <Orientation_nexp> values are the same as values for the Html.orientation command (see below). If the <Orientation_nexp> is not present, the default orientation is determined by the orientation of the device.

Both <ShowStatusBar_lexp> and <Orientation_nexp> are optional; however, a <ShowStatusBar_lexp> must be present in order to specify an <Orientation_nexp>.

Executing a second HTML.OPEN before executing HTML.CLOSE will generate a run-time error.

Html.orientation <nexp>

The value of the <nexp> sets the orientation of screen as follows:

- 1 = Orientation depends upon the sensors.
- 0 = Orientation is forced to Landscape.
- 1 = Orientation is forced to Portrait.
- 2 = Orientation is forced to Reverse Landscape.
- 3 = Orientation is forced to Reverse Portrait.

Html.load.url <file_sexp>

Loads and displays the file specified in the string <file_sexp>. The file may reside on the Internet or on your Android device. In either case, the entire URL must be specified.

The command:

```
HTML.LOAD.URL "http://laughton.com/basic/"
```

will load and display the BASIC! home page.

The command:

```
HTML.LOAD.URL "htmlDemo1.html"
```

will load and display the html file "htmlDemo1.html" residing in BASIC!'s default "data" directory, as set by your "Base Drive" preference. You may also use a fully-qualified pathname. With the default "Base Drive" setting, this command loads the same file:

```
HTML.LOAD.URL "file:///sdcard/rfo-basic/data/htmlDemo1.html"
```

When you tap the BACK key on the originally-loaded page, the HTML viewer will be closed and the BASIC! output console will be displayed. If the page that was originally loaded links to another page and then the BACK key is tapped, it will be up to the BASIC! programmer to decide what to do.

Html.load.string <html_sexp>

Loads and displays the HTML contained in the string expression. The base page for this HTML will be:

```
<pref base drive>/rfo-basic/data/
```

Html.post <url_sexp>, <list_nexp>

Execute a Post command to an Internet location.

<url_sexp> is a string expression giving the url that will accept the Post.

<list_nexp> is a pointer to a string list which contains the Name/Value pairs needed for the Post.

Html.get.datalink <data_svar>

A datalink provides a method for sending a message from an HTML program to the BASIC! programmer.

There are two parts to a datalink in an HTML file:

1. The JavaScript that defines the datalink function
2. The HTML code that calls the datalink function.

The BASIC! Program requires a mechanism for communicating with a website's HTML code.

Html.get.datalink gets the next datalink string from the datalink buffer. If there is no data available then the returned data will be an empty string (""). You should program a loop waiting for data:

```
DO
  HTML.GET.DATALINK data$
UNTIL data$ <> ""
```

The returned data string will always start with a specific set of four characters—three alphabetic characters followed by a colon (":"). These four characters identify the return datalink data type. Most of the type codes are followed by some sort of data. The codes are:

BAK: The user has tapped the BACK key. The data is either "1" or "0".

If the data is "0" then the user tapped BACK in the start screen. Going back is not possible therefore HTML has been closed.

If the data is "1" then going back is possible. The BASIC! programmer should issue the command **Html.go.back** if going back is desired.

LNK: The user has tapped a hyperlink. The linked-to url is returned. The transfer to the new url has not been done. The BASIC! programmer must execute an **Html.load.url** with the returned url (or some other url) for a transfer to occur.

ERR: Some sort of fatal error has occurred. The error condition will be returned. This error code always closes the html engine. The BASIC! output console will be displayed.

FOR: The user has tapped the Submit button on a form with action='FORM' The form name/value pairs are returned.

DNL: The user has clicked a link that requires a download. The download url is supplied. It is up to the BASIC! programmer to do the download.

DAT: The user has performed some action that has caused some JavaScript code to send data to BASIC! by means of the datalink. The JavaScript function for sending the data is:

```
<script type="text/javascript">
  function doDataLink(data) {
    Android.dataLink(data);
  }
</script>
```

Html.go.back

Go back one HTML screen, if possible.

Html.go.forward

Go forward one HTML screen, if possible.

Html.close

Closes the HTML engine and display.

Html.clear.cache

Clears the HTML cache.

Html.clear.history

Clears the HTML history.

Related Commands

Browse <url_sexp>

If <url_sexp> starts with "http..." then the internet site specified by <url_sexp> will be opened and displayed.

If <url_sexp> starts with "[file:///sdcard/...](#)" then the file will be opened by the application, ThinkFree Mobile. The file types that the free version of ThinkFree Mobile can open are ".txt, .doc, .xls, .rtf".

If your Android device does not already have ThinkFree Mobile Viewer on it, you can find it in the apps section of the Google Play Store. There is also a paid "pro" version that manages additional file types.

Note: You can also use the HTML commands to display (and interact with) web pages located on your device or on the web.

Http.post <url_sexp>, <list_nexp>, <result_svar>

Execute a Post command to an Internet location.

<url_sexp> contains the url ("http://....") that will accept the Post.

<list_nexp> is a pointer to a string list which contains the Name/Value pairs needed for the Post.

<result_svar> is where the Post response will be placed.

TCP/IP Sockets

TCP/IP Sockets provide for the transfer of information from one point on the Internet to another. There are two genders of TCP/IP Sockets: Servers and Clients. Clients must talk to Servers. Servers must talk to Clients. Clients cannot talk to Clients. Servers cannot talk to Servers.

Every Client and Server pair have an agreed-upon protocol. This protocol determines who speaks first and the meaning and sequence of the messages that flow between them.

Most people who use a TCP/IP Socket will use a Client Socket to exchange messages with an existing Server with a predefined protocol. One simple example of this is the Sample Program file, **f31_socket_time.bas**. This program uses a TCP/IP client socket to get the current time from one of the many time servers in the USA.

A TCP/IP Server can be set up in BASIC!; however, there are difficulties. The capabilities of individual wireless networks vary. Some wireless networks allow servers. Most do not. Servers can usually be run on WiFi or Ethernet Local Area Networks (LAN).

If you want to set up a Server, the way most likely to work is to establish the Server inside a LAN. You will need to provide Port tunneling (forwarding) from the LAN's external Internal IP to the device's LAN IP. You must be able to program (setup) the LAN router in order to do this.

Clients, whether running inside the Server's LAN or from the Internet, should connect to the LAN's external IP address using the pre-established, tunneled Port. This external or WAN IP can be found using:

```
Graburl ip$, "http://icanhazip.com"
```

This is not the same IP that would be obtained by executing **Socket.myIP** on the server device.

Note: The specified IPs do not have to be in the numeric form. They can be in the name form.

The Sample Program, **f32_tcp_ip_sockets.bas**, demonstrates the socket commands for a Server working in conjunction with a Client. You will need two Android devices to run this program.

TCP/IP Client Socket Commands

Socket.client.connect <server_sexp>, <port_nexp> { , <wait_lexp> }

Create a Client TCP/IP socket and attempt to connect to the Server whose Host Name or IP Address is specified by the Server string expression using the Port specified by Port numeric expression.

The optional "wait" parameter determines if this command waits until a connection is made with the Server. If the parameter is absent or true (non-zero), the command will not return until the connection

has been made or an error is detected. If the Server does not respond, the command should time out after a couple of minutes, but this is not certain.

If the parameter is false (zero), the command completes immediately. Use **Socket.server.status** to determine when the connection is made. If you monitor the socket status, you can set your own time-out policy. You must use the **Socket.client.close** command to stop a connection attempt that has not completed.

Socket.client.status <status_nvar>

Get the current client socket connection status and place the value in the numeric variable <status_nvar>.

- 0 = Nothing going on
- 2 = Connecting
- 3 = Connected

Socket.client.server.ip <svar>

Return the IP of the server that this client is connected to in the string variable.

Socket.client.read.line <line_svar>

Read a line from the previously-connected Server and place the line into the line string variable. The command does not return until the Server sends a line. To avoid an infinite delay waiting for the Server to send a line, the **Socket.client.read.ready** command can be repeatedly executed with timeouts.

Socket.client.read.ready <nvar>

If the previously created Client socket has not received a line for reading by **Socket.client.read.line** then set the return variable <nvar> to zero. Otherwise return a non-zero value.

The **Socket.client.read.line** command does not return until a line has been received from the Server. This command can be used to allow your program to time out if a line has not been received within a pre-determined time span. You can be sure that **Socket.client.read.line** will return with a line of data if **Socket.client.read.ready** returns a non-zero value.

Socket.client.read.file <file_nexp>

Read file data transmitted by the Server and write it to a file. The <file_nexp> is the file index of a file opened for write by **Byte.open write** command. For example:

```
Byte.open w, fw, "image.jpg"
Socket.client.read.file fw
Byte.close fw
```

Socket.client.write.line <line_sexp>

Send the string expression <line_sexp> to the previously-connected Server as UTF-16 characters. End of line characters will be added to the end of the line.

Socket.client.write.bytes <sexp>

Send the string expression, <sexp>, to the previously-connected Server as 8-bit bytes. Each character of the string is sent as a single byte. The string is not encoded. No end-of-line characters are added by BASIC!. If you need a CR or LF character, you must make it part of the string. Note that if

Socket.server.read.line is used to receive these bytes, the **read.line** command will not return until it receives a LF (10, 0x0A) character.

Socket.client.write.file <file_nexp>

Transmit a file to the Server. The <file_nexp> is the file index of a file opened for read by **Byte.open**.

Example:

```
Byte.open r, fr, "image.jpg"
Socket.client.write.file fr
Byte.close fr
```

Socket.client.close

Closes an open client side connection.

TCP/IP Server Socket Commands

Socket.myIP <svar>

Returns the IP of the device in string variable <svar>. If the device has no active IP address, the returned value is the empty string "".

If the device is on a WiFi or Ethernet LAN then the IP returned is the device's LAN IP.

Note: The external or WAN IP can be found using:

```
Graburl ip$, "http://icanhazip.com"
```

Socket.myIP <array\$[]>{ <nvar> }

Returns all active IP addresses of the device in the string array <array\$[]>. If you provide the optional address-count variable <nvar>, it is set to the number of active IP addresses.

If the device has no active IP address, the array has a single element, the empty string "", and the address-count in <nvar> is 0. In this case only, the address-count is not the same as the array length.

Most devices usually have zero or one IP address. It is possible to have more than one. For example, after enabling a WiFi connection, there may still be an active cellular data connection. Normally this connection shuts down after a short time, but in some cases it may remain open.

Socket.server.create <port_nexp>

Establish a Server that will listen to the Port specified by the numeric expression, <port_nexp>.

Socket.server.connect {<wait_lexp>}

Direct the previously created Server to accept a connection from the next client in the queue.

The optional "wait" parameter determines if the command waits until a connection is made with a client. If the parameter is absent or true (non-zero), the command waits for the connection. If the parameter is false (zero), the command completes immediately. Use **Socket.server.status** to determine when the connection is made.

In general, it is safer to set the parameter to false (don't wait) and explicitly monitor the connection's status, since it can avoid a problem if the program exits with no connection made. You must use the **Socket.server.close** command to stop a connection attempt that has not completed.

Socket.server.status <status_nvar>

Get the current server socket connection status and place the value in the numeric variable <status_nvar>.

- 1 = Server socket not created
- 0 = Nothing going on
- 1 = Listening
- 3 = Connected

Socket.server.read.line <svar>

Read a line sent from the previously-connected Client and place the line into the string variable <svar>. The command does not return until the Client sends a line. To avoid an infinite delay waiting for the Client to send a line, the **Socket.server.read.ready** command can be repeatedly executed with timeouts.

Socket.server.read.ready <nvar>

If the previously-connected Client socket has not sent a line for reading by **Socket.server.read.line** then set the return variable <nvar> to zero. Otherwise return a non-zero value.

The **Socket.server.read.line** command does not return until a line has been received from the Client. This command can be used to allow your program to time out if a line has not been received within a pre-determined time span. You can be sure that **Socket.server.read.line** will return with a line of data if it returns a non-zero value.

Socket.server.write.line <line_sexp>

Send the string expression <line_sexp> to the previously-connected Client as UTF-16 characters. End of line characters will be added to the end of the line.

Socket.server.write.bytes <sexp>

Send the string expression, <sexp>, to the previously-connected Client as 8-bit bytes. Each character of the string is sent as a single byte. The string is not encoded. No end of line characters are added by BASIC!. If you need a CR or LF character, you must make it part of the string. Note that if **Socket.client.read.line** is used to receive these bytes, the **read.line** command will not return until it receives a LF (10, 0x0A) character.

Socket.server.write.file <file_nexp>

Transmit a file to the Client. The <file_nexp> is the file index of a file opened for read by **Byte.open**.

Example:

```
Byte.open r, fr, "image.jpg"  
Socket.server.write.file fr  
Byte.close fr
```

Socket.server.read.file <file_nexp>

Read file data transmitted by the Client and write it to a file. The <file_nexp> is the file index of a file opened for write by **Byte.open**. Example:

```
Byte.open w, fw, "image.jpg"  
Socket.server.read.file fw  
Byte.close fw
```

Socket.server.disconnect

Close the connection with the previously-connected Client. A new **Socket.server.connect** can then be executed to connect to the next client in the queue.

Socket.server.close

Close the previously created Server. Any currently connected client will be disconnected.

Socket.server.client.ip <nvar>

Return the IP of the Client currently connected to the Server.

FTP Client

These FTP commands implement a FTP Client

Ftp.open <url_sexp>, <port_nexp>, <user_sexp>, <pw_sexp>

Connects to the specified url and port. Logs onto the server using the specified user name and password. For example:

```
ftp.open "ftp.laughton.com", 21, "basic", "basic"
```

Ftp.close

Disconnects from the FTP server.

Ftp.put <source_sexp>, <destination_sexp>

Uploads specified source file to the specified destination file on connected ftp server.

The source file is relative to the directory, "<pref base drive>/rfo-basic/data/" If you want to upload a BASIC! source file, the file name string would be: "../source/xxxx.bas".

The destination file is relative to the current working directory on the server. If you want to upload to a subdirectory of the current working directory, specify the path to that directory. For example, if there is a subdirectory named "etc" then the filename, "/etc/name" would upload the file into that subdirectory.

Ftp.get <source_sexp>, <destination_sexp>

The source file on the connected ftp server is downloaded to the specified destination file on the Android device.

You can specify a subdirectory in the server source file string.

The destination file path is relative to "<pref base drive>/rfo-basic/data/" If you want to download a BASIC! source file, the path would be, "../source/xxx.bas".

Ftp.dir <list_nvar> {<dirmark_sexp>}

Creates a list of the names of the files and directories in the current working directory and places it in a BASIC! List data structure. A pointer to the new List is returned in the variable <list_nvar>.

A directory is identified by a marker appended to its name. The default marker is the string "(d)". You can change the marker with the optional directory mark parameter <dirmark_sexp>. If you do not want directories to be marked, set <dirmark_sexp> to an empty string, "".

The following code can be used to print the file names in that list:

```
ftp.dir file_list
list.size file_list,size

for i = 1 to size
    list.get file_list,i,name$
    print name$
next i
```

Ftp.cd <new_directory_sexp>

Changes the current working directory to the specified new directory.

Ftp.rename <old_filename_sexp>, <new_filename_sexp>

Renames the specified old filename to the specified new file name.

Ftp.delete <filename_sexp>

Deletes the specified file.

Ftp.rmdir <directory_sexp>

Removes (deletes) the specified directory if and only if that directory is empty.

Ftp.mkdir <directory_sexp>

Creates a new directory of the specified name.

Bluetooth

BASIC! implements Bluetooth in a manner which allows the transfer of data bytes between an Android device and some other device (which may or may not be another Android device).

Before attempting to execute any BASIC! Bluetooth commands, you should use the Android "Settings" Application to enable Bluetooth and pair with any device(s) with which you plan to communicate.

When Bluetooth is opened using the **Bt.open** command, the device goes into the Listen Mode. While in this mode it waits for a device to attempt to connect.

For an active attempt to make a Bluetooth connection, you can use the Connect Mode by successfully executing the **Bt.connect** command. Upon executing the **Bt.connect** command the person running the program is given a list of paired Bluetooth devices and asked. When the user selects a device, BASIC! attempts to connect to it.

You should monitor the state of the Bluetooth using the **Bt.status** command. This command will report states of Listening, Connecting and Connected. Once you receive a "Connected" report, you can proceed to read bytes and write bytes to the connected device.

You can write bytes to a connected device using the **Bt.write** command.

Data is read from the connected device using the **Bt.read.bytes** command; however, before executing **Bt.read.bytes**, you need to find out if there is data to be read. You do this using the **Bt.read.ready** command.

Once connected, you should continue to monitor the status (using **Bt.status**) to ensure that the connected device remains connected.

When you are done with a particular connection or with Bluetooth in general, execute **Bt.close**.

The sample program, f35_bluetooth, is a working example of Bluetooth using two Android devices in a "chat" type application.

Bt.open {0|1}

Opens Bluetooth in Listen Mode. If you do not have Bluetooth enabled (using the Android Settings Application) then the person running the program will be asked whether Bluetooth should be enabled. After **Bt.open** is successfully executed, the code will listen for a device that wants to connect.

The optional parameter determines if BT will listen for a secure or insecure connection. If no parameter is given or if the parameter is 1, then a secure connection request will be listened for. Otherwise, an insecure connection will be listened for. It is not possible to listen for either a secure or insecure

connection with one **Bt.open** command because the Android API requires declaring a specific secure/insecure open.

If **Bt.open** is used in graphics mode (after **Gr.open**), you will need to insert a **Pause 500** statement after the **Bt.open** statement.

Bt.close

Closes any previously opened Bluetooth connection. Bluetooth will automatically be closed when the program execution ends.

Bt.connect {0|1}

Commands BASIC! to connect to a particular device. Executing this command will cause a list of paired devices to be displayed. When one of these devices is selected the **Bt.status** will become "Connecting" until the device has connected.

The optional parameter determines if BT will seek a secure or insecure connection. If no parameter is given or if the parameter is 1, then a secure connection will be requested. Otherwise, an insecure connection will be requested.

Bt.disconnect

Disconnects from the connected Bluetooth device and goes into the Listen status. This avoids having to use **Bt.close** + **Bt.open** to disconnect and wait for a new connection.

Bt.reconnect

This command will attempt to reconnect to a device that was previously connected (during this Run) with **Bt.connect** or a prior **Bt.reconnect**. The command cannot be used to reconnect to a device that was connected following a **Bt.open** or **Bt.disconnect** command (i.e. from the **Listening** status).

You should monitor the Bluetooth status for **Connected** (3) after executing **Bt.reconnect**.

Bt.status {{<connect_var>},{ <name_svar>},{ <address_svar>}}

Gets the current Bluetooth status and places the information in the return variables. The available data are the current connection status (in <connect_var>), and the friendly name and MAC address of your Bluetooth hardware (in <name_svar> and <address_svar>).

All parameters are optional; use commas to indicate omitted parameters (see Optional Parameters).

If the connection status variable <connect_var> is present, it may be either a numeric variable or a string variable. The table shows the possible return values of each type:

Numeric Value	String Value	Meaning
-1	Not enabled	Bluetooth not enabled
0	Idle	Nothing going on

1	Listening	Listening for connection
2	Connecting	Connecting to another device
3	Connected	Connected to another device

If the device name string variable <name_svar> is present, it is set to the friendly device name. If your device has no Bluetooth radio, the string will be empty.

If the address string variable <address_svar> is present, it is set to the MAC address of your Bluetooth hardware, represented as a string of six hex numbers separated by colons: "00:11:22:AA:BB:CC".

Bt.write {<exp> {,|;}} ...

Writes data to the Bluetooth connection.

If the comma (,) separator is used then a comma will be printed between the values of the expressions.

If the semicolon (;) separator is used then nothing will separate the values of the expressions.

If the semicolon is at the end of the line, the output will be transmitted immediately, with no newline character(s) added.

The parameters are the same as the **Print** parameters. This command is essentially a **Print** to the Bluetooth connection, with two differences:

- Only one byte is transmitted for each character; the upper byte is discarded. Binary data and ASCII text are sent correctly, but Unicode characters may not be.
- A line that ends with a semicolon is sent immediately, with no newline character(s) added.

This command with no parameters sends a newline character to the Bluetooth connection.

Bt.read.ready <nvar>

Reports in the numeric variable the number of messages ready to be read. If the value is greater than zero then the messages should be read until the queue is empty.

OnBtReadReady:

Interrupt label that traps the arrival of a message received on the Bluetooth channel (see "Interrupt Labels"). If a Bluetooth message is ready (**Bt.read.ready** would return a non-zero value) BASIC! executes the statements after the **OnBtReady:** label, where you can read and handle the message. When done, execute the **Bt.onReadReady.Resume** command to resume the interrupted program.

Bt.onReadReady.resume

Resumes execution at the point in the program where it was interrupted by the Bluetooth Read Ready event.

Bt.read.bytes <svar>

The next available message is placed into the specified string variable. If there is no message then the string variable will be returned with an empty string ("").

Each message byte is placed in one character of the string; the upper byte of each character is 0. This is similar to **Byte.read.buffer**, which reads binary data from a file into a buffer string.

Bt.device.name <svar>

Returns the name of the connected device in the string variable. A run-time error will be generated if no device (Status <> 3) is connected.

Bt.set.UUID <sexp>

A Universally Unique Identifier (UUID) is a standardized 128-bit format for a string ID used to uniquely identify information. The point of a UUID is that it's big enough that you can select any random 128-bit number and it won't clash with any other number selected similarly. In this case, it's used to uniquely identify your application's Bluetooth service. To get a UUID to use with your application, you can use one of the many random UUID generators on the web.

Many devices have common UUIDs for their particular application. The default BASIC! UUID is the standard Serial Port Profile (SPP) UUID: "00001101-0000-1000-8000-00805F9B34FB".

You can change the default UUID using this command.

Some information about 16 bit and 128 bit UUIDs can be found at:

<http://farwestab.wordpress.com/2011/02/05/some-tips-on-android-and-bluetooth/>

Communication: Phone and Text

Email.send <recipient_sexp>, <subject_sexp>, <body_sexp>

The email message in the Body string expression will be sent to the named recipient with the named subject heading.

MyPhoneNumber <svar>

The phone number of the Android device will be returned in the string variable. If the device is not connected to a cellular network, the returned value will be uncertain.

Phone.call <sexp>

The phone number contained in the string expression will be called. Your device must be connected to a cellular network to make phone calls.

Phone.dial <sexp>

Open the phone dialer app. The phone number contained in the string expression will be displayed in the dialer. Alphabetic characters in the string will be converted to digits, as if the corresponding key of a phone pad had been touched.

Phone.rcv.init

Prepare to detect the state of your phone. If you want to detect phone calls using **Phone.rcv.next**, or you want **Phone.info** to attempt to report signal strength, you must first run this command.

Phone.rcv.init starts a background "listener" task that detects changes in the phone state. There is no command to disable this listener, but it is stopped when your program exits.

Phone.rcv.next <state_nvar>, <number_svar>

The state of the phone will be returned in the state numeric value. A phone number may be returned in the string variable.

State = 0. The phone is idle. The phone number will be an empty string.

State = 1. The phone is ringing. The phone number will be in the string.

State = 2. The phone is off hook. If there is no phone number (an empty string) then an outgoing call is being made. If there is a phone number then an incoming phone call is in progress.

States 1 and 2 will be continuously reported as long the phone is ringing or the phone remains off hook.

Sms.send <number_sexp>, <message_sexp>

The SMS message in the string expression <message_sexp> will be sent to number in the string expression <number_sexp>. This command does not provide any feedback about the sending of the message. The device must be connected to a cellular network to send an SMS message.

Sms.rcv.init

Prepare to intercept received SMS using the sms.rcv.next command.

Sms.rcv.next <svar>

Read the next received SMS message from received SMS message queue in the string variable.

The returned string will contain "@" if there is no SMS message in the queue.

The sms.rcv.init command must be called before the first sms.rcv.next command is executed.

Example:

```
SMS.RCV.INIT
DO
```

```

DO % Loop until SMS received
  PAUSE 5000 % Sleep of 5 seconds
  SMS.RCV.NEXT m$ % Try to get a new message
  UNTIL m$ <> "@" % "@" indicates no new message
  PRINT m$ % Print the new message
UNTIL 0 % Loop forever

```

Time and Timers

Time and TimeZone Commands

The **TimeZone** commands allow you to manage the timezone used by the **Time** command and the **TIME(...)** function. They do not affect the no-parameter **TIME()** function. They affect only your BASIC! program, not any other time-related operation on your device.

See also the time functions **CLOCK()** and **TIME()**.

Time {<time_nexp>}, Year\$, Month\$, Day\$, Hour\$, Minute\$, Second\$, WeekDay, isDST

Returns the current (default) or specified date, time, weekday, and Daylight Saving Time flag in the variables.

You can use the optional first parameter (<time_nexp>) to specify what time to return in the variables. It is a numeric expression number of milliseconds from 12:00:00 AM, January 1, 1970, UTC, as returned by the **TIME()** functions. It may be negative, indicating a time before that date.

The day/date and time are returned as two-digit numeric strings with a leading zero when needed, except Year\$ which is four characters.

The WeekDay is a number from 1 to 7, where 1 means Sunday. You can use it to index an array of day names in your language of choice.

The isDST flag is

- 1 if the current or specified time is in Daylight Saving Time in the current timezone
- 0 if the time is not in Daylight Saving Time (is Standard Time)
- 1 if the system can't tell if the time is in DST

The current timezone is your local timezone unless you change it with the **TimeZone** commands.

All of the return variables are optional. That is, you can omit any of them, but if you want to return only some of them, you need to retain their position by including commas for the omitted return variables.

For example:

```

t = TIME(2001, 2, 3, 4, 5, 6)
Time t, Y$, M$, D$          % sets only the year, month, and day
Time t, Y$, M$, D$, W       % adds the day of the week (7, Saturday)

```

To do the same with the current time, leave out both the first parameter and its comma:

Time ,, day\$,,,, wkday % returns the today's day and weekday

TimeZone.set { <tz_sexp> }

Sets the timezone for your program. If you don't specify a timezone, it is set to the default for your device, which is based on where you are. If you specify a timezone your device does not recognize, it is set to "GMT". (In Android, GMT is exactly the same as UTC).

TimeZone.get <tz_svar>

Returns the current timezone in the string variable. This is the default timezone for your device and location, unless you have changed it with TimeZone.set.

TimeZone.list <tz_list_pointer_nexp>

While timezones are defined by international standards, the only ones that matter to your program are those recognized by your device. This command returns all valid timezone strings, putting them in the list that <tz_list_pointer_nexp> points at. The previous contents of the list are discarded. If the pointer does not specify a valid string list, and the expression is a numeric variable, a new list is created and the variable is set to point to the new list.

Timer Interrupt and Commands

You can set a timer that will interrupt the execution of your program at some set time interval. When the timer expires, BASIC! jumps to the statements following the **OnTimer:** label. When you have done whatever you need to do to handle this Timer event, you use the **Timer.resume** command to resume execution of the program at the point where the timer interrupt occurred.

The timer cannot interrupt an executing command. When the timer expires, the current command is allowed to complete. Then the timer interrupt code after the **OnTimer:** label executes. If the current command takes a long time to finish, it may appear that your timer is late.

The timer cannot interrupt another interrupt. If the timer expires while any interrupt event handler is running, the **OnTimer:** interrupt will be delayed. If the timer expires while the **OnTimer:** interrupt handler is running, the timer event will be lost. The **OnTimer:** interrupt code must exit by running **Timer.resume**, or the timer interrupt can occur only once.

Timer.set <interval_nexp>

Sets a timer that will repeatedly interrupt program execution after the specified time interval. The interval time units are milliseconds. The program must also contain an **OnTimer:** label.

OnTimer:

Interrupt label for the timer interrupt. (See "Interrupt Labels" and "Timer Interrupts and Commands".)

Timer.resume

Resumes execution at the point in the BASIC! program where the **OnTimer:** interrupt occurred.

Timer.clear

Clears the repeating timer. No further timer interrupts will occur.

Sample Code

```
n=0
TIMER.SET 2000

DO
UNTIL n=4
TIMER.CLEAR
PRINT "Timer cleared. No further interrupts."
DO
UNTIL 0

ONTIMER:
n = n + 1
PRINT n*2; " seconds"
TIMER.RESUME
```

Clipboard

Clipboard.get <svar>

Copies the current contents of the clipboard into <svar>

Clipboard.put <sexp>

Places <sexp> into the clipboard.

Encryption

Encrypts and decrypts a string using a supplied password. The encryption algorithm used is "PBESWithMD5AndDES".

Encrypt {<pw_sexp>}, <source_sexp>, <encrypted_svar>

Encrypts the string <source_sexp> using the password <pw_sexp>. The result is placed into the variable <encrypted_svar>.

The password parameter is optional, but its comma is required. Omitting the password is the same as using an empty string, "".

If the source string cannot be encrypted, the result is an empty string (""). You can call the **GETERROR\$()** function to get an error message.

This command is the same as **ENCODE\$("ENCRYPT", <pw_sexp>, <source_sexp>)**.

Decrypt <pw_sexp>, <encrypted_sexp>, <decrypted_svar>

Decrypts the encrypted string <encrypted_sexp> using the password <pw_sexp>. The result is placed in <decrypted_svar>.

The password parameter is optional, but its comma is required. Omitting the password is the same as using an empty string, "".

If the source string cannot be decoded with the specified password, the result is an empty string (""). You can call the **GETERROR\$()** function to get an error message.

This command is the same as **DECODE\$("ENCRYPT", <pw_sexp>, <source_sexp>)**.

Ringer

Android devices support three ringtone modes:

Value:	Meaning:	Behavior
0	Silent	Ringer is silent and does not vibrate
1	Vibrate	Ringer is silent but vibrates
2	Normal	Ringer may be audible and may vibrate

"Normal" behavior depends on other device settings set by the user.

Ringer volume is an integer number from zero to a device-dependent maximum. If the volume is zero the ringer is silent.

NOTE: These are system settings. Any change you make persists after your program ends. You may want to record the original settings and change them back when the program exits.

Ringer.get.mode <nvar>

Returns the current ringtone mode in the numeric variable.

Ringer.set.mode <nexp>

Changes the ringtone mode to the specified value. If the value is not a valid mode, the device mode is not changed.

Ringer.get.volume <vol_nvar> { , <max_nvar> }

Returns the ringer volume level in the numeric variable. Returns the maximum volume setting in <max_nvar>, if <max_nvar> is present.

Ringer.set.volume <nexp>

Changes the ringer volume to the specified value. If the value is less than zero, volume is set to zero. If the value is greater than the device-specific maximum, the volume is set to the maximum level.

String Operations

Split and **Join** are complementary operations. **Split** separates a string into parts and put the parts in an array. **Join** builds a string by combining the elements of an array.

See also the various String Functions.

Join <source_array\$[]>, <result_svar> {, <separator_sexp>{, <wrapper_sexp>}}

Join.all <source_array\$[]>, <result_svar> {, <separator_sexp>{, <wrapper_sexp>}}

The elements of the <source_array\$[]> are joined together as a single string in the <result_svar>. By default, the source elements are joined with nothing between them or around them.

You may specify optional modifiers that add characters to the string. Copies of the separator string <separator_sexp> are written between source elements. Copies of the wrapper string <wrapper_sexp> are placed before and after the rest of the result string.

The **Join** command omits any empty source elements. The **Join.all** command includes all source elements in the result string, even if they are empty. There is no difference between the two commands unless you specify a non-empty separator string. **Join.all** places copies of the separator between all of the elements, including the empty ones.

An example of an operation that uses both separators and wrappers is a CSV string, for "comma-separated values".

```
InnerPlanets$ = "Mercury Venus Earth Mars"
SPLIT IP[], InnerPlanets$
JOIN IP[], PlanetsCSV$, "\",\"", "\""
PRINT PlanetsCSV$
```

This prints the string "Mercury", "Venus", "Earth", "Mars" (including all of the quotes). The separator puts the ", " between planet names, and the wrapper puts the " at the beginning and end of the string.

Split <result_array\$[]>, <sexp> {, <test_sexp>}

Split.all <result_array\$[]>, <sexp> {, <test_sexp>}

Splits the source string <sexp> into multiple strings and place them into <result_array\$[]>. The array is specified without an index. If the array exists, it is overwritten. Otherwise a new array is created. The result is always a one-dimensional array.

The string is split at each location where <test_sexp> occurs. The <test_sexp> occurrences are removed from the result strings. The <text_sexp> parameter is optional; if it is not given, the string is split on whitespace. Omitting the parameter is equivalent to specifying "\\s+".

If the beginning of the source string matches the test string, the first element of the result array will be an empty string. This differs from the **WORD\$()** function, which strips leading and trailing occurrences of the test string from the source string before splitting.

Two adjacent occurrences of the test expression in the source expression result in an empty element somewhere in the result array. The **Split** command discards these empty strings if they occur at the end of the result array. To keep these trailing empty strings, use the **Split.all** command.

Example:

```
string$ = "a:b:c:d"
delimiter$ = ":"
SPLIT result$[], string$, delimiter$

ARRAY.LENGTH length, result$[]
FOR i = 1 TO length
PRINT result$[i] + " ";
NEXT i
PRINT ""
```

Prints: a b c d

Note: The <test_sexp> is actually a Regular Expression. If you are not getting the results that you expect from the <test_sexp> then you should examine the rules for Regular Expressions at:

<http://developer.android.com/reference/java/util/regex/Pattern.html>

Speech Conversion

Text To Speech

Your program can synthesize speech from text, either for immediate playback with the **TTS.speak** command or to create a sound file with **TTS.speak.toFile**.

Your device may come with the text-to-speech engine already enabled and configured, or you may need to set it up yourself in the Android Settings application. The details vary between different devices and versions of Android. Typically, the menu navigation looks like one of these:

Settings → Voice input & output → Text-to-speech settings
Settings → Language & input → Text-to-speech output

Unless you set an output language in the text-to-speech settings, the speech generated will be spoken in the current default language of the device. The menu path for setting the default language usually looks like one of these:

Settings → Language and keyboard → Select language
Settings → Language & input → Language

Most speech engines limit the number of characters they are able to speak. The limit is not the same for all devices or all speech engines, but it is typically around 4000 characters. If you exceed the limit, most engines fail silently: you don't get an error message, but you don't get any speech output, either.

TTS.init

This command must be executed before speaking.

TTS.speak <sexp> { <wait_lexp> }

Speaks the string expression. The statement does not return until the string has been fully spoken, unless the optional "wait" parameter is present and evaluates to false (numeric 0). Spoken expressions cannot overlap. A second **TTS.speak** (or a **TTS.speak.toFile**) will wait for the speech from an earlier **TTS.speak** to finish, even if the "wait" flag was false.

TTS.speak.toFile <sexp> { <path_sexp> }

Converts the string expression to speech and writes it into a wav file. You can specify the name and location of the file with the optional "path" parameter. The default path is "<pref base drive>/rfo-basic/data/tts.wav". The statement does not return until the speech synthesis is complete, but there is no guarantee the file-write is finished. If a previous **TTS.speak** is still speaking, this statement will not start until that speech completes.

TTS.stop

Waits for any outstanding speech to finish, then releases Android's text-to-speech engine. Following **TTS.stop**, if you want to run **TTS.speak** or **TTS.speak.toFile** again, you will have to run **TTS.init** again.

Speech To Text (Voice Recognition)

The Voice Recognition function on Android uses Google Servers to perform the recognition. This means that you must be connected to the Internet and logged into your Google account for this feature to work.

There are two commands for Speech to Text: **STT.listen** and **STT.results**.

STT.listen starts the voice recognition process with a dialog box. **STT.results** reports the interpretation of the voice with a list of strings.

The Speech to Text procedures are different for Graphics Mode, HTML mode and simple Console Output mode.

STT.listen {<prompt_sexp>}

Start the voice recognize process by displaying a "Speak Now" dialog box. The optional prompt string expression <prompt_sexp> sets the dialog box's prompt. If you do not provide the prompt parameter, the default prompt "BASIC! Speech To Text" is used.

Begin speaking when the dialog box appears.

The recognition will stop when there is a pause in the speaking.

STT.results should be executed next.

Note: **STT.listen** is *not* to be used in HTML mode.

STT.results <string_list_ptr_nexp>

The command must not be executed until after a **STT.listen** is executed (unless in HTML mode).

The recognizer returns several variations of what it thinks it heard as a list of strings. The first string in the list is the best guess.

The strings are written into the list that <string_list_ptr_nexp> points to. The previous contents of the list are discarded. If the pointer does not specify a valid string list, and the expression is a numeric variable, a new list is created and the variable is set to point to the new list.

Console Mode

The following code illustrates the command in Output Console (not HTML mode and not Graphics mode):

```
PRINT "Starting Recognizer"
STT.LISTEN
STT.RESULTS theList
LIST.SIZE theList, theSize
FOR k = 1 TO theSize
    LIST.GET theList, k, theText$
    PRINT theText$
NEXT k
END
```

Graphics Mode

This command sequence is to be used in graphics mode. Graphics mode exists after **Gr.open** and before **Gr.close**. (Note: Graphics mode is temporarily exited after **Gr.front 0**. Use the Console Mode if you have called **Gr.front 0**).

The primary difference is that **Gr.render** *must* be called after **STT.listen** and before **STT.results**.

```
PRINT "Starting Recognizer"
STT.LISTEN
GR.RENDER
STT.RESULTS theList
LIST.SIZE theList, theSize
FOR k =1 TO theSize
    LIST.GET theList, k, theText$
    PRINT theText$
NEXT k
END
```

HTML Mode

This command sequence is used while in HTML mode. HTML mode exists after **HTML.open** and before **HTML.close**.

The primary difference is that the **STT.listen** command is *not* used in HTML mode. The **STT.listen** function is performed by means of an HTML datalink sending back the string "STT". The sending of "STT" by means of the datalink causes the Speak Now dialog box to be displayed.

When the datalink "STT" string is received by the BASIC! program, the **STT.results** command can be executed normally as it will contain the recognized text.

The sample file, f37_html_demo.bas, along with the associated html file, htmlDemo1.html (located in "rfo-basic/data/") demonstrates the use of voice recognition in HTML mode.

Information About Your Android Device

Device Command Overview

You can get information about your Android device with the **Device** command:

- The Device Brand, Device Model, Device Type, and OS
- The Language and Locale
- The PhoneType, PhoneNumber, and DeviceID
- The SN, MCC/MNC, and Network Provider stored on the SIM, if there is one.

The **Device** command has two forms that differ only in the type of the parameter, which determines the format of the returned data. Both forms return the same information, as shown in this table:

Key	Values	Meaning	Example (from emulator)
Brand	Any string	Brand name assigned by device manufacturer	generic
Model	Any string	Model identifier assigned by device manufacturer	sdk
Device	Any string	Device identifier assigned by device manufacturer	generic
Product	Any string	Product identifier assigned by device manufacturer	sdk
OS	OS Version	Android operating system version number	4.1.2
Language	Language name	Default language of this device	English
Locale	Locale code	Default locale code, typically language and country	en_US
PhoneType	GSM, CDMA, SIP, or None	Type of phone radio in this device	GSM

PhoneNumber	String of digits	Phone number registered to this device, if any	15555215554
DeviceID	String of digits	The unique device ID, such as the IMEI	000000000000000
SIM SN	String of digits or Not available	Serial number of the SIM card, if one is present and it is accessible	8901410321111851 0720
SIM MCC/MNC	String of digits or Not available	The "numeric name" of the provider of the SIM, if present and accessible	310260
SIM Provider	Name string or Not available	The name of the provider of the SIM, if present and accessible	Android

The last six items access your device's telephony system and SIM card. If your device has no telephone, or BASIC! does not have permission to access them, the fields are set to neutral values: "None", "Not available", or a string of "0" characters.

In addition, there are convenience commands to retrieve only the Locale or the Language.

The information returned by **Device** is static. To get dynamic information, use the **Phone.Info** command.

Device <svar>

Returns information about your Android device in the string variable <svar>. Each item has this form:

```
key = value
```

The names `key` and `value` refer to the first two columns of the table in the **Device** overview above. The items are placed in a single string, separated by newline characters. Formatted this way, if you **Print** the string it is displayed with one item on each line. You can separate the individual items with **Split**:

```
DEVICE info$ % all info in one string
SPLIT info$[], info$, "\n" % each array element is one item, "<k> = <v>"
SPLIT lang_line$[], info$[6], " = " % split the Language item, two-element array
lang$ = lang_line$[2] % lang$ is the language name, like "English"
```

Device <nexp>|<nvar>

Returns information about your Android device in a Bundle. If you provide a variable that is not a valid Bundle pointer, the command creates a new Bundle and returns the Bundle pointer in your variable. Otherwise it writes into the Bundle your variable or expression points to.

The Bundle keys are shown in the first column of the table in the **Device** overview above.

```
DEVICE info % all info in a Bundle
BUNDLE.GET info, "Language", lang$ % lang$ is the language name, like "English"
```

Device.Language <svar>

Returns the default language of the device as a human-readable string value in string variable <svar>.

This convenience shortcut returns the same value as the "Language" key of the Bundle returned by the numeric form of the **Device** command.

Device.Locale <svar>

Returns the default locale code of the device in standard Locale format, typically including the language and country codes.

This convenience shortcut returns the same value as the "Locale" key of the Bundle returned by the numeric form of the **Device** command.

Phone.info <nexp>|<nvar>

Returns information about the telephony radio in your Android device, if it has one. The information is placed in a Bundle. If you provide a variable that is not a valid Bundle pointer, the command creates a new Bundle and returns the Bundle pointer in your variable. Otherwise it writes into the Bundle your variable or expression points to.

The Bundle keys and possible values are in the table below. Each entry's type is either N (Numeric) or S (String).

Key	Type	Values	Meaning	Example
PhoneType	S	GSM, CDMA, SIP, or None	Type of phone radio in this device	GSM
NetworkType	S	GPRS, EDGE, UMTS, CDMA, EVDOrev0, EVDOrevA, 1xRTT, HSDPA, HSUPA, HSPA, iDen, EDV0revB, LTE, EHRPD, HSPAP+, or Unknown	Network type of the current data connection	LTE

The **PhoneType** is the same as that returned by the **Device** command. It is static.

If the **PhoneType** is **GSM** and the phone is registered to a network, the **Phone.info** command also returns the following items in the Bundle:

Key	Type	Values	Meaning	Example
CID	N	A positive number or -1 if CID is unknown	GSM Cell ID	342298497
LAC	N	A positive number or -1 if LAC is unknown	GSM Location Area Code	11090
MCC/MNC	S	String of 5 or 6 decimal digits	The "numeric name" of the registered network operator	310260
Operator	N	A name string	The name of the operator of the registered network	T-Mobile

The "numeric name" is made up of the Mobile Country Code (MCC) and Mobile Network Code (MNC).

If the **PhoneType** is **CDMA** and the phone is registered to a network, the **Phone.info** command also returns the following items in the Bundle:

Key	Type	Values	Meaning
BaseID	N	A positive number or -1 if BaseID is unknown	CDMA base station identification number
NetworkID	N	A positive number or -1 if NetworkID is unknown	CDMA network identification number
SystemID	N	A positive number or -1 if SystemID is unknown	CDMA system identification number

If your program has executed **Phone.rcv.init**, then **Phone.info** may be able to report the strength of the signal connecting your phone to the cell tower. If the signal strength is available, **Phone.info** will try to report it in one or two of the following bundle keys (the first three are mutually exclusive):

Key	Type	Values	Meaning
SignalLevel	N	A positive number 0 - 4	General measure of signal quality as shown in status bar. Higher is better.
GsmSignal	N	A positive number, 0 - 31 or 99 if unknown	"SignalLevel" unavailable, phone type is GSM, get GSM level instead.
CdmaDbm	N	A negative number, typically -90 (strong) to -105 (weak)	"SignalLevel" unavailable, phone type is CDMA, get raw power level in dBm instead.
SignalASU	N	0 - 31, 99 (most) 0 - 97, 99 (LTE)	"Arbitrary Strength Units", range depends on network type.

This information is not available on some Android devices, depending on the device manufacturer and your wireless carrier.

Screen rotation, size[], realsize[], density

Returns information about your screen.

- Provide numeric variables to get the rotation and density of the screen.
- Provide numeric array variables to get the "application size" and "real size" of the screen.
A size is returned as two values, width first and height second, in a two-element array. If the array exists, it is overwritten. Otherwise a new array is created.

All parameters are optional. Use commas to indicate omitted parameters (see Optional Parameters).

rotation: The current orientation of your screen relative to the "natural" orientation of your device. The natural orientation may be portrait or landscape, as defined by the manufacturer. The return value is a number from 0 to 3. Multiply by 90 to get the rotation in degrees clockwise.

size[]: The size of the screen in pixels available for applications. This excludes system decorations. The width and height values reflect the current screen orientation.

realSize[]: The current real size of the screen in pixels, including system decorations. NOTE: this value is available only on devices running Android version 4.2 or later. On other devices, the values will be the same as in the **size[]** array parameter.

density: A standardized Android density value in dots per inch (dpi), usually 120, 160, or 240 dpi. This is not necessarily the real physical density of the screen. This value never changes.

Screen.rotation <nvar>

Returns a number in the <nvar> parameter representing the current orientation of your screen relative to the "natural" orientation of your device. The natural orientation may be portrait or landscape, as defined by the manufacturer. The return value is a number from 0 to 3. Multiply by 90 to get the rotation in degrees clockwise.

Screen.size, size[], realSize[], density

Returns information about the size and density of your screen.

- You may provide numeric array variables to get the "application size" and "real size" of the screen. A size is returned as two values, width first and height second, in a two-element array. If the array exists, it is overwritten. Otherwise a new array is created.
- Provide a simple numeric variable to get the density of the screen.

All parameters are optional. Use commas to indicate omitted parameters (see Optional Parameters).

size[]: The size of the screen in pixels available for applications. This excludes system decorations. The width and height values reflect the current screen orientation.

realSize[]: The current real size of the screen in pixels, including system decorations. NOTE: this value is available only on devices running Android version 4.2 or later. On other devices, the values will be the same as in the **size[]** array parameter.

density: A standardized Android density value in dots per inch (dpi), usually 120, 160, or 240 dpi. This is not necessarily the real physical density of the screen. This value never changes.

If your program is running in Graphics mode, "**SCREEN.SIZE xy[], , dens**" returns the same values as "**GR.SCREEN x, y, dens**". Unlike **Gr.screen**, **Screen.size** also works in Console and HTML modes.

WiFi.info {{<SSID_svar>},{ <BSSID_svar>},{ <MAC_svar>},{ <IP_var>},{ <speed_nvar>}}

Gets information about the current Wi-Fi connection and places it in the return variables.

All of the parameters are optional; use commas to indicate omitted parameters (see Optional Parameters). The table shows the available data:

Variable	Type	Returned Data	Format
SSID	String	SSID of current 802.11 network	"name" or hex digits (see below)

BSSID	String	BSSID of current access point	xx:xx:xx:xx:xx:xx (MAC address)
MAC	String	MAC address of your WiFi	xx:xx:xx:xx:xx:xx
IP	Numeric or String	IP address of your WiFi	Number or octets (see below)
speed	Numeric	Current link speed in Mbps	Number

Format notes:

- SSID: If the network is named, the name is returned, surrounded by double quotes. Otherwise the returned name is a string of hex digits.
- IP: If you provide a numeric variable, your Wi-Fi IP address is returned as a single number. If you provide a string variable, the number is converted to a standard four-octet string. For example, the string format 10.70.13.143 is the same IP address as the number -1887287798 (hex 8f82460a).

Running in the Background

Home

The **HOME** command does exactly what tapping the HOME key would do. The Home Screen is displayed while the BASIC! program continues to run in the background.

OnBackground:

Interrupt label that traps changes in the Background/Foreground state (see "Interrupt Labels"). BASIC! executes the statements following the **OnBackground:** label until it reaches a **Background.resume** command. The **Background()** function should be used to determine the new state.

Background.resume

Resumes execution at the point in the BASIC! program where the **OnBackground:** interrupt occurred.

WakeLock <code_nexp>{, <flags_nexp>}

The **WakeLock** command modifies the system screen timeout function. The Code parameter <code_nexp> may be one of five values. Values 1 through 4 modify the screen timeout in various ways. Code value 5 releases the WakeLock and restores the system screen timeout function.

Code	WakeLock Type	CPU	Screen	Keyboard Light
<u>1</u>	Partial WakeLock	On*	Off	Off
<u>2</u>	Screen Dim	On	Dim	Off
<u>3</u>	Screen Bright	On	Bright	Off
<u>4</u>	Full WakeLock	On	Bright	Bright
5	No WakeLock	Off	Off	Off

** If you hold a partial WakeLock, the CPU will continue to run, regardless of any timers and even after the user taps the power button. In all other WakeLocks, the CPU will run, but the user can still put the device to sleep using the power button.*

You can use the optional Flags parameter <flags_nexp> to change the screen behavior, as shown in the table below. If the WakeLock Type is Partial WakeLock (Code 1), the Flags parameter is ignored.

Value	Flag(s) Set	Meaning
1	Acquire causes wakeup	Turn screen on when wakelock is acquired.
2	On after release	Reset screen timeout when wakelock is released.
3	Both	
Other	Neither	

Use the WakeLock only when you really need it. Acquiring a WakeLock increases power usage and decreases battery life. The WakeLock is always released when the program stops running.

One of the common uses for **WakeLock** would be in a music player that needs to keep the music playing after the system screen timeout interval. Implementing this requires that BASIC! be kept running. One way to do this is to put BASIC! into an infinite loop:

```
Audio.load n, "B5b.mp3"
Audio.play n
WakeLock 1
Do
  Pause 30000
Until 0
```

The screen will turn off when the system screen timeout expires but the music will continue to play.

WifiLock <code_nexp>

The **WifiLock** command allows your program to keep the WiFi connection awake when a time-out would normally turn it off. The <code_nexp> may be one of four values. Values 1 through 3 acquire a WifiLock, changing the way WiFi behaves when the screen turns off. Code value 4 releases the WifiLock and restores the normal timeout behavior.

Code:	WiFiLock Type	WiFi Operation When Screen is Off
1	Scan Only	WiFi kept active, but only operation is initiating scan and reporting scan results
2	Full	WiFi behaves normally
3	Full High-Performance	WiFi operates at high performance with minimum packet loss and low latency*
4	No WifiLock	WiFi turns off (may be settable on some devices)

*Full Hi-Perf mode is not available on all Android devices. Devices running Android 3.0.x or earlier, and devices without the necessary hardware, will run in Full mode instead.

Use **WifiLock** only when you really need it. Acquiring a WifiLock increases power usage and decreases battery life. The WifiLock is always released when the program stops running.

Miscellaneous Commands

Headset <state_nvar>, <type_svar>, <mic_nvar>

Reports if there is a headset plugged into your device, and returns data about the headset. The parameters are all names of variables that receive the data:

- <state_nvar>: 1.0 if a headset is plugged in, 0.0 if no headset is plugged in, and -1.0 if unknown.
- <type_svar>: A string describing the device type of the last headset known to your device.
- <mic_nvar>: 1.0 if the headset has a microphone, 0.0 if the headset does not have a microphone, and -1.0 if unknown.

If you plug in or unplug a headset, new information becomes available. Your program must run the **Headset** command again to get the update.

Notify <title_sexp>, <subtitle_sexp>, <alert_sexp>, <wait_lexp>

This command will cause a Notify object to be placed in the Notify (Status) bar. The Notify object displays the BASIC! app icon and the <alert_sexp> text. The user taps the Notify object to open the notification window. Your program's notification displays the <title_sexp> and <subtitle_sexp> text.

The code snippet and screenshots shown below demonstrate the placement of the parameter strings.

If <wait_lexp> is not zero (true), then the execution of the BASIC! program will be suspended until the user taps the Notify object. If the value is zero (false), the BASIC! program will continue executing.

The Notify object will be removed when the user taps the object, or when the program exits.

```
Print "Executing Notify"
Notify "BASIC! Notify", "Tap to resume running program",~
"BASIC! Notify Alert", 1
! Execution is suspended and waiting for user to tap the Notify Object
Print "Notified"
```





Note: the icon that appears in the Notify object will be the icon for the application in user-built apk.

Pause <ticks_nexp>

Stops the execution of the BASIC! program for <ticks_nexp> milliseconds. One millisecond = 1/1000 of a second. Pause 1000 will pause the program for one second. A pause can not be interrupted.

An infinite loop can be a very useful construct in your programs. For example, you may use it to wait for the user to tap a control on the screen. A tight spin loop keeps BASIC! very busy doing nothing. A **Pause**, even a short one, reduces the load on the CPU and the drain on the battery. Depending on your application, you may want to add a **Pause** to the loop to conserve battery power:

```
DO : PAUSE 50 : UNTIL x <> 0
```

Swap <nvar_a>|<svar_a>, <nvar_b>|<svar_b>

The values in "a" and "b" numeric or string variables are swapped.

Tone <frequency_nexp>, <duration_nexp> {, <duration_chk_lexp>

Plays a tone of the specified frequency in hertz (cycles per second) for the specified duration in milliseconds.

The duration produced does not exactly match the specified duration. If you need to get an exact duration, experiment.

Each Android device has a minimum tone duration. By default, if you specify a duration less than this minimum, you get a run-time error message giving the minimum for your device. However, you can suppress the check by setting the optional duration check flag <duration_chk_lexp> to 0 (false). If you do this, the result you get depends on your device. You will not get a run-time error message, but you may or may not get the tone you expect.

Vibrate <pattern_array[{<start>,<length>}]>,<nexp>

The vibrate command causes the device to vibrate in the specified pattern. The pattern is held in a numeric array (pattern_array[]) or array segment (pattern_array[start,length]).

The pattern is of the form: pause-time, on-time, ..., pause-time, on-time. The values for pause-time and on-time are durations in milliseconds. The pattern may be of any length. There must be at least two values to get a single buzz because the first value is a pause.

If <nexp> = -1 then the pattern will play once and not repeat.

If <nexp> = 0 then the pattern will continue to play over and over again until the program ends.

If <nexp> > 0 then the pattern play will be cancelled.

See the sample program, **f21_sos.bas**, for an example of **Vibrate**.

VolKeys

Certain keys and buttons have special meaning to your Android device. By default, BASIC! propagates these special keycodes to the Android system, even if your program detects them. So, for example, when you press the "Volume Up" button, your program can catch it (see **INKEY\$**), but you still see the Volume Control window open on your screen, and your audio gets louder.

The **VolKeys.off** and **VolKeys.on** commands let you control this behavior for five keys. These keys may be on the Android device or on a headset plugged into the device.

Key Name (Android docs)	Key Code (decimal)	Usual Action
VOLUME_UP	24	Increase speaker volume
VOLUME_DOWN	25	Decrease speaker volume
VOLUME_MUTE	164	Mute the speaker (Android 3.0 or later)
MUTE	91	Mute the microphone
HEADSETHOOK	79	Hang up a call, stop media playback

VolKeys.off

Disables the usual action of the keys listed in the VolKeys table, above. Your program can still detect these keypresses, but BASIC! does not pass the events on to the Android system.

VolKeys.on

Enables the usual action of the keys listed in the VolKeys table, above. This is the default setting when your BASIC! program starts.

SQLITE

Overview

The Android operating system provides the ability to create, maintain and access SQLite databases. SQLite implements a [self-contained](#), [serverless](#), [zero-configuration](#), [transactional](#) SQL database engine. SQLite is the [most widely deployed](#) SQL database engine in the world. The full details about SQLite can be found at the [SQLite Home Page \(http://www.sqlite.org/\)](http://www.sqlite.org/).

An excellent online tutorial on SQL can be found at [www.w3schools.com](http://www.w3schools.com/sql/default.asp) (<http://www.w3schools.com/sql/default.asp>).

Database files will be created on the base drive (usually the SD card) in the directory, "<pref base drive>/rfo-basic/databases/".

SQLITE Commands

Sql.open <DB_pointer_nvar>, <DB_name_sexp>

Opens a database for access. If the database does not exist, it will be created.

<DB_pointer_nvar> is a pointer to the newly opened database. This value will be set by the sql.open command operation. <DB_pointer_nvar> is used in subsequent database commands and should not be altered.

<DB_name_sexp> is the filename used to hold this database. The base reference directory is "<pref base drive>/com.rfo.basic/databases/". If <DB_name_sexp> = ":memory:" then a temporary database will be created in memory.

Note: You may have more than one database opened at same time. Each opened database must have its own, distinct pointer.

Sql.close <DB_pointer_nvar>

Closes a previously opened database. <DB_pointer_nvar> will be set to zero. The variable may then be reused in another sql.open command. You should always close an opened database when you are done with it. Not closing a database can reduce the amount of memory available on your Android device.

Sql.new_table <DB_pointer_nvar>, <table_name_sexp>, C1\$, C2\$, ...,CN\$

A single database may contain many tables. A table is made of rows of data. A row of data consists of columns of values. Each value column has a column name associated with it.

This command creates a new table with the name <table_name_sexp> in the referenced opened database. The column names for that table are defined by the following: C1\$, C2\$, ..., CN\$. At least one column name is required. You may create as many column names as you need.

BASIC! always adds a Row Index Column named "_id" to every table. The value in this Row Index Column is automatically incremented by one for each new row inserted. This gives each row in the table a unique identifier. This identifier can be used to connect information in one table to another table. For example, the _id value for customer information in a customer table can be used to link specific orders to specific customers in an outstanding order database.

Sql.drop_table <DB_pointer_nvar>, <table_name_sexp>

The table named <table_name_sexp> in the opened database pointed to by <DB_pointer_nvar> will be dropped from the database if the table exists.

Sql.insert <DB_pointer_nvar>, <table_name_sexp>, C1\$, V1\$, C2\$, V2\$, ..., CN\$, VN\$

Inserts a new row of data columns and values into a table in a previously opened database.

The <table_name_sexp> is the name of the table into which the data is to be inserted. All newly inserted rows are inserted after the last, existing row of the table.

C1\$, V1\$, C2\$, V2\$, ..., CN\$, VN\$: The column name and value pairs for the new row. These parameters must be in pairs. The column names must match the column names used to create the table. Note that the values are all strings. When you need a numeric value for a column, use the BASIC! STR\$(n) to convert the number into a string. You can also use the BASIC! FORMAT\$(pattern\$, N) to create a formatted number for a value. (The Values-as-strings requirement is a BASIC! SQL Interface requirement, not a SQLite requirement. While SQLite, itself, stores all values as strings, it provides transparent conversions to other data types. I have chosen not to complicate the interface with access to these SQLite conversions since BASIC! provides its own conversion capabilities.)

Sql.query <cursor_nvar>, <DB_pointer_nvar>, <table_name_sexp>, <columns_sexp> {, <where_sexp> {, <order_sexp>}}

Queries a table of a previously-opened database for some specific data. The command returns a Cursor named <Cursor_nvar> to be used in stepping through Query results.

The <columns_sexp> is a string expression with a list of the names of the columns to be returned. The column names must be separated by commas. An example is Columns\$ = "First_name, Last_name, Sex, Age". If you want to get the automatically incremented Row Index Column then include the "_id" column name in your column list. Columns may be listed in any order. The column order used in the query will be the order in which the rows are returned.

The optional <where_sexp> is an SQL expression string used to select which rows to return. In general, an SQL expression is of the form <Column Name> <operator> <Value>. For example, Where\$ = "First_name = 'John' " Note that the Value must be contained in single quotes. Full details about the SQL expressions can be found [here](#). If the Where parameter is omitted, all rows will be returned.

The optional <order_sexp> specifies the order in which the rows are to be returned. It identifies the column upon which the output rows are to be sorted. It also specifies whether the rows are to be sorted in ascending (ASC) or descending (DESC) order. For example, Order\$ = "Last_Name ASC" would return the rows sorted by Last_Name from A to Z. If the Order parameter is omitted, the rows are not sorted.

If the Order parameter is present, the Where parameter must be present. If you want to return all rows, just set Where\$ = ""

Sql.query.length <length_nvar>, <cursor_nvar>

Report the number of records returned by a previous Query command, Given the Cursor returned by a Query, the command writes the number of records into <length_nvar>. This command cannot be used after all of the data has been read.

Sql.query.position <position_nvar>, <cursor_nvar>

Report the record number most recently read using the Cursor of a Query command. Given the Cursor returned by a Query, the command writes the position of the Cursor into <Position_nvar>. Before the first Next command, the Position is 0. It is incremented by each Next command. A Next command after the last row is read sets its Done variable to true and resets the Cursor to 0. The Cursor can no longer be used, and this command can no longer be used with that Cursor.

Sql.next <done_lvar>, <cursor_nvar>{, <cv_svars>}

Using the Cursor generated by a previous Query command (**Sql.query** or **Sql.raw_query**), step to the next row of data returned by the Query and retrieve the data.

<done_lvar> is a Boolean variable that signals when the last row of the Query data has been read.

<cursor_nvar> is a numeric variable that holds the Cursor pointer returned by a Query command. You may have more than one Cursor open at a time.

<cv_svars> is an optional set of column value string variables that return data from the database to your program. The Cursor carries the values from the table columns listed in the Query. **Sql.next** retrieves one row from the Cursor as string values and writes them into the <cv_svars>. If any of your columns are numeric, you can use the BASIC! **VAL(str\$)** function to convert the strings to numbers.

When this command reads a row of data, it sets the Done flag <done_lvar> to false (0.0). If it finds no data to read, it changes the Done flag to true (1.0) and resets the cursor variable <cursor_nvar> to zero. The Cursor can no longer be used for **Sql.next** operations. The cursor variable may be used with another Cursor from a different Query.

In its simplest form, <cv_svars> is a comma-separated list of string variable names. Each variable receives the data of one column. If there are more variables than columns, the excess variables are left unchanged. If there are more columns than variables, the excess data are discarded.

```
SQL.NEXT done, cursor, cv1$, cv2$, cv3$    % get data from up to three columns
```

The last (or only) variable may be a string array name with no index(es):

```
SQL.NEXT done, cursor, cv$[]              % get data from ALL available columns
```

The data from any column(s) that are not written to string variables are written into the array. If no column data are written to the array, the array has one element, an empty string "". If the variable names an array that already exists, it is overwritten.

If the last (or only) <cv_svars> variable is an array, you may also add (after another comma) a numeric variable <ncol_nvar>. This variable receives the total number of columns in the cursor. Note that this is not necessarily the same as the size of the array.

```
SQL.NEXT done, cursor, cv$[], nCols       % report number of columns available
```

The full specification for this command, including the optional array and column count, is as follows:

```
Sql.next <done_lvar>, <cursor_nvar> {, svar}... {, array$[] {, <ncol_nvar>}}
```

Sql.delete <DB_pointer_nvar>, <table_name_sexp>{<where_sexp>{<count_nvar>}}

From the named table of a previously opened database, delete rows selected by the conditions established by the Where string expression. The Count variable reports the number of rows deleted.

The formation of the Where string is exactly the same as described in the **Sql.query** command. Both Where and Count are optional. If the Where string is omitted all rows are deleted, and the Count variable must be omitted, too.

Sql.update <DB_pointer_nvar>, <table_name_sexp>, C1\$, V1\$, C2\$, V2\$,...,CN\$, VN\${: <where_sexp>}

In the named table of a previously opened database, change column values in specific rows selected by the Where\$ parameter <where_sexp>. The C\$,V\$ parameters must be in pairs. The colon character terminates the C\$,V\$ list and must precede the Where\$ in this command. The Where\$ parameter and preceding colon are optional.

BASIC! also uses the colon character to separate multiple commands on a single line. The use of a colon in this command conflicts with that feature. Use caution when using both together.

If you put a colon on a line after this command, the preprocessor **always** assumes the colon is part of the command and not a command separator. If you are not certain of the outcome, the safest action is to put the **Sql.update** command on a line by itself, or at the end of a multi-command line.

Sql.exec <DB_pointer_nvar>, <command_sexp>

Execute ANY non-query SQL command string ("CREATE TABLE", "DELETE", "INSERT", etc.) using a previously opened database.

Sql.raw_query <cursor_nvar>, <DB_pointer_nvar>, <query_sexp>

Execute ANY SQL Query command using a previously opened database and return a Cursor for the results.

Graphics

Introduction

The Graphics Screen and Graphics Mode

Graphics are displayed on a new screen that is different from the BASIC! Text Output Screen. The Text Output Screen still exists and can still be written to. You can be returned to the text screen using the BACK key or by having the program execute the **Gr.front** command.

The **Gr.open** command opens the graphics screen and puts BASIC! into the graphics mode. BASIC! must be in graphics mode before any other graphics commands can be executed. Attempting to execute any graphics command when not in the graphics mode will result in a run-time error. The **Gr.close** command closes the graphics screen and turns off graphics mode. The graphics mode automatically turns off when the BACK key or MENU key is tapped. BASIC! will continue to run after the BACK key or MENU key is tapped when in graphics mode but the Output Console will be shown.

The BASIC! Output Console is hidden when the graphics screen is being displayed. No run-time error messages will be observable. A haptic feedback alert signals a run-time error. This haptic feedback will

be a distinct, short buzz. Tap the BACK key to close the Graphics Screen upon feeling this alert. The error messages can then read from the BASIC! Output Console.

Use the **Gr.front** command to swap the front-most screen between the Output Console and the graphics screen.

Commands that use a new window or screen to interact with the user (**Input**, **Select** and others) may be used in graphics mode.

When your program ends, the graphics screen will be closed. If you want to keep the graphics screen showing, use a long pause or an infinite loop to keep the program from ending:

```
! Stay running to keep the graphics screen showing
do
until 0
```

Depending on your application, you may want to add a **Pause** to the loop to conserve battery power. Tap the BACK key to break out of the infinite loop. The BACK key ends your program unless you trap it with the **OnBackKey:** interrupt label.

Display Lists

Each command that draws a graphical object (line, circle, text, etc.) places that object on a list called the Object List. The command returns the object's Object Number. This Object Number, or Object Pointer, is the object's position in the Object List. This Object Number can be used to change the object on the fly. You can change the parameters of any object in the Object List with the **Gr.modify** command. This feature allows you easily to create animations without the overhead of having to recreate every object in the Object List.

To draw graphical objects on the graphics screen, BASIC! uses a Display List. The Display List contains pointers to graphical objects on the Object List. Each time a graphical object is added to the Object List, its Object Number is also added to the Display List. Objects are drawn on the screen in the order in which they appear in the Display List. The Display List objects are not visible on the screen until the **Gr.render** command is called.

You may use the **Gr.NewDL** command to replace the current Display List with a custom display list array. This custom display list array may contain some or all of the Object Numbers in the Object List.

One use for custom display lists is to change the Z order of the objects. In other words you can use this feature to change which objects will be drawn on top of other objects.

See the Sample Program file, f24_newdl, for a working example of **Gr.NewDL**.

Drawing Coordinates

The size and location of an object drawn on the screen are specified in pixels. The coordinates of the pixel at the upper-left corner of the screen are x = 0 (horizontal position) and y = 0 (vertical position). Coordinate values increase from left to right and from top to bottom of the screen.

Coordinates are measured with respect to the physical screen, not to anything on it. If you choose to show the Android Status Bar, anything you draw at the top of the screen is covered by the Status Bar.

Drawing into Bitmaps

You can draw into bitmaps in addition to drawing directly to the screen. You notify BASIC! that you want to start drawing into a bitmap instead of the screen with the **Gr.bitmap.drawinto.start** command. This puts BASIC! into the draw-into-bitmap mode. All draw commands issued while in this mode will draw directly into the bitmap. The objects drawn in this mode will not be placed into the Object List. The Object Number returned by a draw command while in this mode is invalid and should not be used for any purpose including **Gr.modify**.

The draw-into-bitmap mode is ended by the **Gr.bitmap.drawinto.end** command. Subsequent draw commands will place the objects on the Object List and object numbers in the Display List for rendering on the screen. If you wish to display the drawn-into bitmap on the screen, issue a **Gr.bitmap.draw** command for that bitmap. The drawn-into bitmap may be drawn at any time before, during or after the draw-into process.

Colors

BASIC! colors consist of a mixture of Red, Green, and Blue. Each component has a numerical value ranging from 0 to 255. Black occurs when all three values are zero. White occurs when all three values are 255. Solid Red occurs with a Redvalue of 255 while Blue and Green are zero.

Colors also have what is called an Alpha Channel. The Alpha Channel describes the level of opaqueness of the color. An Alpha value of 255 is totally opaque. No object of any color can show through an object with an Alpha value of 255. An Alpha value of zero renders the object invisible.

Paints

BASIC! holds drawing information such as color, font size, style and so forth, in Paint objects. The Paint objects are stored in a Paint List. The last created Paint object (the "Current Paint") is associated with a graphical object when a draw command is executed. The Paint tells the renderer (see **Gr.render**) how to draw the graphical object. The same Paint may be attached to many graphical objects so they will all be drawn with the same color, style, etc.

Basic usage

You can ignore Paints. This can keep your graphics programming simpler.

Each command that changes a drawing setting (**Gr.color**, **Gr.text.size**, etc.) affects everything you draw from that point on, until you execute another such command.

Advanced usage

You can control the Paint objects used to draw graphical objects. The extra complexity allows you to create special effects that are not otherwise possible. To use these effects, you must understand the Paint List and the Current Paint.

Each command that changes a drawing setting (**Gr.color**, **Gr.text.size**, etc.) creates or modifies a Paint. Each of these commands has an optional "Paint pointer" parameter. If you don't specify which Paint to use, the command first copies the Current Paint and then modifies it to make a new Paint, which then becomes the Current Paint. If you do specify which Paint to use, the command modifies only the specified Paint, leaving the Current Paint unchanged.

The Current Paint is always the last Paint on the Paint List. If you specify the Paint object, the pointer values -1 and 0 are special.

Paint pointer value -1 means the Current Paint. Using -1 is the same as omitting the Paint pointer parameter.

Paint pointer value 0 refers to a "working Paint". You can change it as often as you like without generating a new Paint. Paint 0 can not be attached to a graphical object. To make it useful, you must copy it (**Gr.paint.copy**) to another location in the Paint List, or to the Current Paint.

You can get a pointer to the current Paint with the **Gr.paint.get** command. You can get a pointer to the Paint associated with any graphical object by using the "paint" tag with **Gr.get.value** command. You can assign that Paint to any other graphical object by using the "paint" tag with **Gr.modify** command.

Paints can not be individually deleted. You may delete all Paint objects, along with all graphics objects, with **Gr.cls**.

The commands **Gr.paint.copy** and **Gr.paint.reset** operate directly on Paint objects.

Style

Most graphics objects are made up of two parts: an outline and the center. There is a subtle but important difference in the rules governing how each part is drawn. The two parts are controlled by the style setting of the Paint object, set by the style parameter of the **Gr.color** command.

Note: Gr.Point and Gr.Line are not affected by style. Only the STROKE part (outline) is drawn.

STROKE

If you specify STROKE (style 0), only the outline is drawn. The center of the shape is not colored. The width of the outline is controlled by the stroke weight setting of the Paint, set by the **Gr.stroke** command. If the stroke weight is 0 or 1 (they behave the same way), the outline is drawn exactly on the coordinates you provide. For example, if the shape is a rectangle:

left: The left-most edge. All pixels with x-coordinate **left** are colored.

top: The upper-most edge. All pixels with y-coordinate **top** are colored.

right: The right-most edge. All pixels with x-coord **right** are colored.

bottom: The lower-most edge. All pixels with y-coord **bottom** are colored.

This is probably what you expect.

When you increase the stroke weight, the lines get wider, but the centers of the lines do not change. Additional lines of pixels are colored on both sides of the outline. Increasing the stroke weight generally increases the area of the shape, making it larger than the dimensions you specify.

This may not be what you expect. In some drawing systems, increasing the line width grows the line inward only, toward the center of the shape, so the shape does not get any bigger.

FILL

If you specify FILL (style 1), the center of the shape is filled as if it had an outline. That is, the center of the object is colored, but the pixels colored by STROKE may be left uncolored. The area that is colored depends on the coordinates of the shape. For example, consider a rectangle:

left: The left-most edge. All pixels with x-coordinate **left** are colored.

top: The upper-most edge. All pixels with y-coordinate **top** are colored.

right: One more than the right-most edge. All pixels with x-coord **right - 1** are colored.

bottom: One more than the lower-most edge. Pixels with y-coord **bottom - 1** are colored.

left and **top** values are "inclusive": pixels with x-coordinate **left** are colored. Pixels with y-coordinate **top** are colored.

right and **bottom** values are "exclusive": pixels with x-coordinate **right** are **not** colored. Pixels with y-coordinate **bottom** are **not** colored.

This is not what most people expect, but it is consistent with many operations in Java programming.

STROKE and FILL

If you specify STROKE_AND_FILL (style 2), both parts of the shape are drawn and superimposed. That is, the outline is drawn as described in STROKE, above, and the object is filled in as described in FILL. Because both parts are the same color, and there are never uncolored pixels between the STROKE lines and the FILL area, the effect is to draw a single solid shape. Note that increasing the stroke weight generally makes the shape bigger than the dimensions you specify.

Hardware-accelerated Graphics

Many Android devices since 3.0 (Honeycomb) support hardware acceleration of some graphical operations. An app that can use the hardware Graphics Processor (GPU) may run significantly faster than one that cannot use the GPU. Some of BASIC!'s graphical operations do not work with hardware-acceleration, so it is disabled by default. You can turn it on with the **Graphic Acceleration** item of the **Editor->Menu->Preferences** screen.

If you enable accelerated graphics, test your app thoroughly, comparing it to what you see with acceleration off. If you see blurring, missing objects, or other problems, leave acceleration disabled.

Graphics Setup Commands

Gr.open {{alpha}}{{, red}}{{, green}}{{, blue}}{{, <ShowStatusBar_lexp>}}{{<Orientation_nexp>}}

Opens the Graphics Screen and puts BASIC! into Graphics Mode. The color values become the background color of the graphics screen. The default color is opaque white (255,255,255,255).

All parameters are optional; use commas to indicate omitted parameters (see Optional Parameters).

Each of the four color components is a numeric expression with a value from 0 through 255. If a value is outside of this range, only the last eight bits of the value are used; for example, 257 and 1025 are the same as 1. If any color parameter is omitted, it is set to 255.

The Status Bar will be shown on the graphics screen if the <ShowStatusBar_lexp> is true (not zero). If the <ShowStatusBar_lexp> is not present, the Status Bar will not be shown.

The orientation upon opening graphics will be determined by the <Orientation_nexp> value.

<Orientation_nexp> values are the same as values for the **Gr.orientation** command (see below). If the <Orientation_nexp> is not present, the default orientation is Landscape.

Gr.color {{alpha}}{{, red}}{{, green}}{{, blue}}{{, style}}{{, paint}}

Sets the color and style for drawing objects. There are two ways to use this command.

- *Basic usage:* ignore the optional <paint> parameter. The new color and style will be used for whatever graphical objects are subsequently drawn until the next **Gr.color** command is executed.
- *Advanced usage:* The "basic usage" of this command always creates a new Paint. If you prefer, you can use the <paint> parameter to specify an existing Paint. The **Gr.color** command sets the color and style of that Paint, changing the appearance of any graphical object to which it is attached. The current Paint is not changed. See "Paints *Advanced Usage*" above and the example below.

All of the parameters are optional. If a color component or the style is omitted, that component is left unchanged. For example, **Gr.color ,,0** sets only green to 0, leaving alpha, red, blue, and style as they were. Use commas to indicate omitted parameters (see Optional Parameters).

Each of the four color components (alpha, red, green, blue) is a numeric expression with a value from 0 through 255. If a value is outside of this range, only the last eight bits of the value are used; for example, 257 and 1025 are both the same as 1.

The style parameter, is a numeric expression that determines the stroking and filling of objects. The effect of this parameter is explained in detail in the "Style" sections, see above. The possible values for <style_nexp> are shown in this table:

Value	Meaning	Description
0	STROKE	Geometry and text drawn with this style will be stroked (outlined), respecting the stroke-related fields on the paint.
1	FILL	Geometry and text drawn with this style will be filled, ignoring all

		stroke-related settings in the paint.
2	STROKE_AND_FILL	Geometry and text drawn with this style will be filled and stroked at the same time, respecting the stroke-related fields on the paint.

If you specify a value other than -1, 0, 1, or 2, then the style is set to 2. If you specify a style of -1, the style is left unchanged, just as if the style parameter were omitted. If you never set a style, the default value is 1, FILL.

You can change the stroke weight with commands such as **Gr.set.stroke** (see below) and the various text style commands.

Example:

```
GR.OPEN
! basic usage
GR.COLOR ,0,0,255,2           % opaque blue, stroke and fill
GR.RECT r1, 50,50,100,100     % draw two squares
GR.RECT r2, 100,100,150,150
GR.COLOR 128,255,0,0         % half-transparent red
GR.RECT r3, 75,75,125,125     % draw an overlapping square
GR.RENDER : PAUSE 2000
! advanced usage
GR.GET.VALUE r1, "paint", p   % get index of first Paint
GR.COLOR 255,0,255,0,,p      % change that Paint's color to opaque green
GR.RENDER : PAUSE 2000       % both r1 and r2 change
GR.RECT r4, 125,125,175,175   % use current Paint, unchanged
GR.RENDER : PAUSE 2000       % still draws half-transparent red
GR.CLOSE : END
```

Gr.set.antialias {{<lexp>}}{<paint_nexp>}}

Turns antialiasing on or off on objects drawn after this command is issued:

- If the value of the antialias setting parameter <lexp> is false (0), AntiAlias is turned off.
- If the parameter value is true (not zero), AntiAlias is turned on.
- If the parameter is omitted, the AntiAlias setting is toggled.

AntiAlias should generally be on. It is on by default.

AntiAlias must be off to draw single-pixel pixels and single-pixel-wide horizontal and vertical lines.

You may use the optional Paint pointer parameter <paint_nexp> to specify a Paint object to modify. Normally this parameter is omitted. See **Gr.color**, *Advanced usage* for more information.

Gr.set.stroke {{<nexp>}}{<paint_nexp>}}

Sets the line width of objects drawn after this command is issued. The <nexp> value must be >=0. Zero produces the thinnest line and is the default stroke value.

The thinnest horizontal lines and vertical lines will be two pixels wide if AntiAlias is on. Turn AntiAlias off to draw single-pixel-wide horizontal and vertical lines.

Pixels drawn by the **Gr.set.pixels** command will be drawn as a 2x2 matrix if AntiAlias is on. To draw single-pixel pixels, set AntiAlias off and set the stroke = 0.

You may use the optional Paint pointer parameter <paint_nexp> to specify a Paint object to modify. Normally this parameter is omitted. See **Gr.color**, *Advanced usage* for more information.

Gr.orientation <nexp>

The value of the <nexp> sets the orientation of screen as follows:

- 1 = Orientation depends upon the sensors.
- 0 = Orientation is forced to Landscape.
- 1 = Orientation is forced to Portrait.
- 2 = Orientation is forced to Reverse Landscape.
- 3 = Orientation is forced to Reverse Portrait.

You can monitor changes in orientation by reading the screen width and height using the the **Gr.screen** or **Screen** commands.

Gr.statusbar {<height_nvar>} {, <showing_lvar>}

Returns information about the Status Bar. If the **height** variable <height_nvar> is present, it is set to the nominal height of the Status Bar. If the **showing** flag <showing_lvar> is present, it is set to **0** (false, not showing) or **1** (true, showing) based on how Graphics Mode was opened.

The parameters are both optional. If you omit the first parameter but use the second, you must keep the comma.

Gr.statusbar.show <nexp>

This command has been deprecated. To show the status bar on the graphics screen, use the optional fifth parameter in **Gr.open**.

Gr.render

This command displays all the objects that are listed in the current working Display List. It is not necessary to have a **Pause** command after a **Gr.render**. The **Gr.render** command will not complete until the contents of the Display List have been fully displayed.

Gr.render always waits until the next screen refresh. Most Android devices refresh the screen 60 times per second; your device may be faster or slower. Therefore, if you execute two consecutive **Gr.render** commands, there will be a delay of 16.7 milliseconds (on most devices) between the two commands.

For smooth animation, try to avoid doing more than 16.7 ms of work between **Gr.render** commands, to achieve the maximum refresh rate. This is not a lot of time for a BASIC! program, so you may have to settle for a lower frame rate. However, there is no benefit to trying to render more often than 16.7 ms.

If BASIC! is running in the background (see **Background()** function and **Home** command), **Gr.render** will not execute. It will pause your program until you return BASIC! to the foreground.

Gr.screen width, height{, density }

Returns the screen's width and height, and optionally its density, in the numeric variables. The density, in dots per inch (dpi), is a standardized Android density value (usually 120, 160, or 240 dpi), and not necessarily the real physical density of the screen.

If a **Gr.orientation** command changes the orientation, the width and height values from a previous **Gr.screen** command are invalid.

Android's orientation-change animation takes time. You may need to wait for a second or so after **Gr.open** or **Gr.orientation** before executing **Gr.screen**, otherwise the width and height values may be set before the orientation change is complete.

Gr.screen returns a subset of the information returned by the newer **Screen** command.

Gr.scale x_factor, y_factor

Scale all drawing commands by the numeric x and y scale factors. This command is provided to allow you to draw in a device-independent manner and then scale the drawing to the actual size of the screen that your program is running on. For example:

```
! Set the device independent sizes
di_height = 480
di_width = 800

! Get the actual width and height
gr.open      % defaults: white, no status bar, landscape
gr.screen actual_w, actual_h

! Calculate the scale factors
scale_width = actual_w / di_width
scale_height = actual_h / di_height

! Set the scale
gr.scale scale_width, scale_height
```

Now, start drawing based upon `di_height` and `di_width`. The drawings will be scaled to fit the device running the program.

Gr.cls

Clears the graphics screen. Deletes all previously drawn objects; all existing object references are invalid. Deletes all existing Paints and resets all **Gr.color** or **Gr.text** {size|align|bold|strike|underline|skew} settings. Disposes of the current Object List and Display List and creates a new Initial Display List.

Note: bitmaps are not deleted. They will not be drawn because no graphical objects point to them, but the bitmaps still exist. Variables that point to them remain valid.

The **Gr.render** command must be called to make the cleared screen visible to the user.

Gr.close

Closes the opened graphics mode. The program will continue to run. The graphics screen will be removed. The text output screen will be displayed.

Gr.front flag

Determines whether the graphics screen or the Output Console will be the front-most screen. If flag = 0, the Output Console will be the front-most screen and seen by the user. If flag <> 0, the graphics screen will be the front-most screen and seen by the user.

One use for this command is to display the Output Console to the user while in graphics mode. Use **Gr.front 0** to show text output and **Gr.front 1** to switch back to the graphics screen.

Note: When the Output Console is in front of the graphics screen, you can still draw (but not render) onto the graphics screen. The **Gr.front 1** must be executed before any **Gr.render**.

Print commands will continue to print to the Output Console even while the graphic screen is in front.

Gr.brightness <nexp>

Sets the brightness of the graphics screen. The value of the numeric expression should be between 0.01 (darkest) and 1.00 (brightest).

Graphical Object Creation Commands

These commands create graphical objects and add them to the Object List, also adding their Object Numbers to the Display List. You create each object with parameters that describe what to draw and where. Once it is created, you can read back its parameters by name with the **Gr.get.value** command. You can change any parameter with the **Gr.modify** command. The parameters you can modify are listed with each command's description. Along with the parameters listed with each command, every graphical object has two other modifiable parameters, "paint" and "alpha". See the **Gr.modify** and **Gr.paint.get** command descriptions for more details.

There are three commands that create graphical objects that are not in this section: **Gr.text.draw**, **Gr.bitmap.draw**, and **Gr.clip**.

Gr.point <obj_nvar>, x, y

Creates a point object. The point will be located at (x,y). The <obj_nvar> returns the Object List object number for this point. This object will not be visible until the **Gr.render** command is called.

The appearance of the point object is affected by the current stroke weight and the AntiAlias setting. The object is rendered as a square, centered on (x,y) and as big as the current stroke. If AntiAlias is on, it will blur the point, making it larger and dimmer. To color a single pixel, use **Gr.set.stroke 0** and **Gr.set.antialias 0**.

The **Gr.modify** parameters for **Gr.point** are: "x" and "y".

Gr.line <obj_nvar>, x1, y1, x2, y2

Creates a line object. The line will start at (x1,y1) and end at (x2,y2). The <obj_nvar> returns the Object List object number for this line. This object will not be visible until the **Gr.render** command is called.

The thinnest horizontal lines and vertical lines are drawn with **Gr.set.stroke 0**. These lines will be two pixels wide if AntiAlias is on. Turn AntiAlias off to draw single-pixel wide horizontal and vertical lines.

The **Gr.modify** parameters for **Gr.line** are: "x1", "y1", "x2" and "y2".

Gr.rect <obj_nvar>, left, top, right, bottom

Creates a rectangle object. The rectangle will be located within the bounds of the parameters. The rectangle will or will not be filled depending upon the **Gr.color** style parameter. The <obj_nvar> returns the Object List object number for this rectangle. This object will not be visible until the **Gr.render** command is called.

The **Gr.modify** parameters for **Gr.rect** are: "left", "top", "right" and "bottom".

Gr.oval <obj_nvar>, left, top, right, bottom

Creates an oval-shaped object. The oval will be located within the bounds of the parameters. The oval will or will not be filled depending upon the **Gr.color** style parameter. The <obj_nvar> returns the Object List object number for this oval. This object will not be visible until the **Gr.render** command is called.

The **Gr.modify** parameters for **Gr.oval** are: "left", "top", "right" and "bottom".

Gr.arc <obj_nvar>, left, top, right, bottom, start_angle, sweep_angle, fill_mode

Creates an arc-shaped object. The arc will be created within the rectangle described by the parameters. It will start at the specified start_angle and sweep clockwise through the specified sweep_angle. The angle values are in degrees.

The effect of the fill_mode parameter depends on the **Gr.color** style parameter:

- Style 0, fill_mode false: Only the arc is drawn.
- Style 0, fill_mode true: The arc is drawn with lines connecting each endpoint to the center of the bounding rectangle. The resulting closed figure is not filled.
- Style non-0, fill_mode false: The endpoints of the arc are connected by a single straight line. The resulting figure is filled.
- Style non-0, fill_mode true: The arc is drawn with lines connecting each endpoint to the center of the bounding rectangle. The resulting closed figure is filled.

The <obj_nvar> returns the Object List object number for this arc. This object will not be visible until the **Gr.render** command is called.

The **Gr.modify** parameters for **Gr.arc** are: "left", "top", "right", "bottom", "start_angle", "sweep_angle" and "fill_mode". The value for "fill_mode" is either false (0) or true (not 0).

Gr.circle <obj_nvar>, x, y, radius

Creates a circle object. The circle will be created with the given radius around the designated center (x,y) coordinates. The circle will or will not be filled depending upon the **Gr.color** style parameter. The <obj_nvar> returns the Object List object number for this circle. This object will not be visible until the **Gr.render** command is called.

The **Gr.modify** parameters for **Gr.circle** are "x", "y", and "radius".

Gr.set.pixels <obj_nvar>, pixels[{<start>,<length>}] {x,y}

Inserts an array of pixel points into the Object List. The array (pixels[]) or array segment (pixels[start,length]) contains pairs of x and y coordinates for each pixel. The pixels[] array or array segment may be any size but must have an even number of elements.

If the optional x,y expression pair is present, the values will be added to each of the x and y coordinates of the array. This provides the ability to move the pixel array around the screen. The default values for the x,y pair is 0,0. Negative values for the x,y pair are valid.

Pixels will be drawn as 2x2 matrix pixels if AntiAlias is on and the stroke = 0. To draw single-pixel pixels, set AntiAlias off and set the stroke = 0. AntiAlias is on by default.

The <obj_nvar> returns the Object List object number for the object. The pixels will not be visible until the **Gr.render** command is called.

The **Gr.modify** parameters for this command are "x" and "y".

In addition to modify, the individual elements of the pixel array can be changed on the fly. For example:

```
Pixels[3] = 120  
Pixels[4] = 200
```

will cause the second pixel to be located at x = 120, y = 200 at the next rendering.

Gr.poly <obj_nvar>, list_pointer {x, y}

Creates an object that draws a closed polygon of any number of sides. The <obj_nvar> returns the Object List object number for this polygon. This object will not be visible until the next **Gr.render**.

The list_pointer is an expression that points to a List data structure. The list contains x,y coordinate pairs. The first coordinate pair defines the point at which the polygon drawing starts. Each subsequent coordinate pair defines a line drawn from the previous coordinate pair to this coordinate pair. A final line drawn from the last point back to the first closes the polygon.

If the optional x,y expression pair is present, the values will be added to each of the x and y coordinates of the list. This provides the ability to move the polygon array around the screen. The default x,y pair is 0,0. Negative values for x and y are valid.

The polygon line width, line color, alpha and fill are determined by previous **Gr.color** and **Gr.set.stroke** commands just like any other drawn object. These attributes are owned by the **poly** object, not by the list. If you use the same list in different **Gr.poly** commands, the color, stroke, etc., may be different.

You can change the polygon (add, delete, or move points) by directly manipulating the list with **List** commands. You can change to a different list of points using **Gr.Modify** with "list" as the tag parameter. Changes are not visible until the **Gr.render** command is called.

When you create a polygon with **Gr.poly** or attach a new list with **Gr.modify**, the list must have an even number of values and at least two coordinate pairs (four values). These rules are enforced with run-time errors. The rules cannot be enforced when you modify the list with **List** commands. Instead, if you have an odd number of coordinates, the last is ignored. If you have only one point, **Gr.render** draws nothing.

The **Gr.modify** parameters are "x", "y" and "list".

See the Sample Program file, f30_poly, for working examples of **Gr.poly**.

Groups

You can put graphical objects into groups. A group is a list of graphical objects. When you perform certain operation on a group, the operation is performed on each object in the group.

You group graphical objects by creating a Group Object on the Display List. You use the group by putting its object number in a graphics command where you would use any other graphical object number.

In this version of BASIC!, you can use a Group Object in these commands:

- **Gr.move:** moves all of the objects by the same x and y amounts
- **Gr.hide:** hides all of the objects
- **Gr.show:** shows (unhides) all of the objects
- **Gr.show.toggle:** any objects that are showing will be hidden, and any objects that are hidden will be shown.

You use graphics commands to act on the objects in the group's list. You use the **List** commands to act on the list: add objects, count the objects, clear the list, and so on.

Try running this example. Watch as the top circle moves to the right, then the top two, and finally the top three, as circles are added one-by-one to the list attached to the group.

```
GR.OPEN ,,,,1 : GR.COLOR ,255,0,0,2
GR.CIRCLE c1,100,100,40 : GR.CIRCLE c2,100,200,40
GR.CIRCLE c3,100,300,40 : GR.CIRCLE c4,100,400,40
GR.RENDER : PAUSE 1000           % draw four red circles

GR.GROUP g, c1                  % create a group with one circle
GR.GET.VALUE g, "list", gList   % get the group's list of objects
GR.MOVE g, 0, 50                % move whole group 0 up/down, 50 right
GR.RENDER : PAUSE 1000          % only one circle moves
```

```

LIST.ADD gList, c2           % add another circle to the group's list
GR.MOVE g, 0, 50
GR.RENDER : PAUSE 1000      % two circles move

LIST.ADD gList, c2           % add another circle to the group's list
GR.MOVE g, 0, 50            % three circles move
GR.RENDER : PAUSE 1000

GR.CLOSE : END

```

Gr.group <object_number_nvar>{, <obj_nexp>}...

Creates a group of graphical objects. All of the numeric expressions <obj_nexp> must evaluate to valid graphical object numbers. The object numbers are put in a list and attached to the group.

The <object_number_nvar> returns the Object List object number for the group.

The **Gr.modify** parameter is "list".

Gr.group.list <object_number_nvar>, <list_ptr_nexp>

Creates a group from a list of graphical objects. The List is assumed to contain valid graphical object numbers, but it is not checked. The list is simply attached to the group.

The list pointer parameter <list_ptr_nexp> is optional. If you provide an expression that evaluates to a valid List pointer, the List that the pointer addresses supplies the graphical objects that are put in the group. Otherwise, the group is empty. If you provide a numeric variable that does not already point to a list, the variable is set to point to the group's empty list.

The <object_number_nvar> returns the Object List object number for the group.

The **Gr.modify** parameter is "list".

Gr.group.getDL <object_number_nvar>

Creates a group from the current Display List. The Display List is copied to a new list that is attached to the group.

The <object_number_nvar> returns the Object List object number for the group.

The **Gr.modify** parameter is "list".

Gr.group.newDL <object_number_nvar>

Replaces the existing Display List with a new list read from the specified group.

The <object_number_nvar> returns the Object List object number for the group.

The **Gr.modify** parameter is "list".

Hide and Show Commands

Gr.hide <object_number_nexp>

Hides the object with the specified Object Number. If the Object is a Group, all of the Graphical Objects in the Group are hidden. This change will not be visible until the **Gr.render** command is called.

Gr.show <object_number_nexp>

Shows (unhides) the object with the specified Object Number. If the Object is a Group, all of the Graphical Objects in the Group are shown. This change will not be visible until the **Gr.render** command is called.

Gr.show.toggle <object_number_nexp>

Toggles visibility of the object with the specified Object Number. If it is hidden, it will be shown. If it is shown, it will be hidden. If the Object is a Group, all of the Graphical Objects in the Group are toggled. This change will not be visible until the **Gr.render** command is called.

Touch Query Commands

If the user touches the screen and then moves the finger without lifting the finger from the screen, the motion can be tracked by repeatedly calling on the touch query commands. This will allow you to program the dragging of graphical objects around the screen. The Sample Program, f23_breakout.bas, illustrates this with the code that moves the paddle.

The OnGrTouch: label can be used optionally to interrupt your program when a new touch is detected.

The touch commands report on one- or two-finger touches on the screen. If the two fingers cross each other on the x-axis then touch and touch2 will swap.

Gr.touch touched, x, y

Tests for a touch on the graphics screen. If the screen is being touched, Touched is returned as true (not 0) with the (x,y) coordinates of the touch. If the screen is not currently touched, Touched returns false (0) with the (x,y) coordinates of the last previous touch. If the screen has never been touched, the x and y variables are left unchanged. The command continues to return true as long as the screen remains touched.

If you want to detect a single short tap, after detecting the touch, you should loop until touched is false.

```
DO
  GR.TOUCH touched, x, y
UNTIL touched

! Touch detected, now wait for
! finger lifted
DO
  GR.TOUCH touched, x, y
UNTIL !touched
```

The returned values are relative to the actual screen size. If you have scaled the screen then you need to similarly scale the returned parameters. If the parameters that you used in **Gr.scale** were `scale_x` and `scale_y` (**Gr.scale scale_x, scale_y**) then divide the returned x and y by those same values.

```
GR.TOUCH touched, x, y
Xscaled = x / scale_x
Yscaled = y / scale_y
```

Gr.bounded.touch touched, left, top, right, bottom

The Touched parameter will be returned true (not zero) if the user has touched the screen within the rectangle defined by the left, top, right, bottom parameters. If the screen has not been touched or has been touched outside of the bounding rectangle, the touched parameter will be return as false (zero). The command will continue to return true as long as the screen remains touched and the touch is within the bounding rectangle.

The bounding rectangle parameters are for the actual screen size. If you have scaled the screen then you need to similarly scale the bounding rectangle parameters. If the parameters that you used in **Gr.scale** were `scale_x` and `scale_y` (**Gr.scale scale_x, scale_y**) then divide left and right by `scale_x` and divide top and bottom by `scale_y`.

Gr.touch2 touched, x, y

The same as **Gr.touch** except that it reports on second simultaneous touch of the screen.

Gr.bounded.touch2 touched, left, top, right, bottom

The same as **Gr.bounded.touch** except that it reports on second simultaneous touch of the screen.

OnGrTouch:

Interrupt label that traps any touch on the Graphics screen (see "Interrupt Labels"). BASIC! executes the statements following the **OnGrTouch:** label until it reaches a **Gr.onGrTouch.resume** command.

To detect touches on the Output Console (not in Graphics mode), use **OnConsoleTouch:**.

Gr.onGrTouch.resume

Resumes execution at the point in the BASIC! program where the **OnGrTouch:** interrupt occurred.

Text Commands

Overview

Gr.text.draw is the only **text** command that creates a graphical object. The other **text** commands set attributes of text yet to be drawn or report measurements of text.

Each command that sets a text attribute (the **Gr.text.align**, **bold**, **size**, **skew**, **strike**, and **underline** commands, as well as **Gr.text.setfont** and **typeface**) has an optional "Paint pointer" parameter that may be used to specify a Paint object to modify. Normally this parameter is omitted, and the command sets a text attribute for all text objects drawn after the command is executed. For using the Paint pointer, see "Paints *Advanced Usage*", above.

Gr.text.height returns information about how text would be drawn. It does not measure a drawn text object, but uses information from the current Paint.

Gr.text.width and **Gr.get.textbounds** commands can return information about how text would be drawn or how a text object was actually drawn. If used to measure the text of a string expression, they use information from the current Paint. If used to measure the text of a text object that was already drawn, they use information from the text object.

Gr.text.align {{<type_nexp>},{<paint_nexp>}}

Align the text relative to the (x,y) coordinates given in the **Gr.text.draw** command.

type values: 1 = Left, 2 = Center, 3 = Right.

You may use the optional Paint pointer parameter <paint_nexp> to specify a Paint object to modify. Normally this parameter is omitted. See **Gr.color**, *Advanced usage* for more information.

Gr.text.bold {{<lexp>},{<paint_nexp>}}

Turns bold on or off on text objects drawn after this command is issued:

- If the value of the bold parameter <lexp> is false (0), text bold is turned off.
- If the parameter value is true (not zero), makes text appear bold.
- If the parameter is omitted, the bold setting is toggled.

If the color fill parameter is 0, only the outline of the bold text will be shown. If fill <>0, the text outline will be filled.

This is a "fake bold", simulated by graphical techniques. It may not look the same as text drawn after setting "Bold" style with **Gr.text.setfont** or **Gr.text.typeface**.

You may use the optional Paint pointer parameter <paint_nexp> to specify a Paint object to modify. Normally this parameter is omitted. See **Gr.color**, *Advanced usage* for more information.

Gr.text.size {{<size_nexp>},{<paint_nexp>}}

Specifies the size of the text in pixels. The size <nexp> sets the nominal height of the characters. This height is large enough to include the top of characters with ascenders, like "h", and the bottom of characters with descenders, like "y". Character width is scaled proportionately to the height.

You may use the optional Paint pointer parameter <paint_nexp> to specify a Paint object to modify. Normally this parameter is omitted. See **Gr.color**, *Advanced usage* for more information.

Gr.text.skew {{<skew_nexp>},{<paint_nexp>}}

Skews the text to give an italic effect. Negative values of <nexp> skew the bottom of the text left. This makes the text lean forward. Positive values do the opposite. Traditional italics can be best imitated with <nexp> = -0.25.

You may use the optional Paint pointer parameter <paint_nexp> to specify a Paint object to modify. Normally this parameter is omitted. See **Gr.color**, *Advanced usage* for more information.

Gr.text.strike {{<lexp>}}{,<paint_nexp>}}

Turns overstrike on or off on text objects drawn after this command is issued:

- If the value of the strike parameter <lexp> is false (0), text strike is turned off.
- If the parameter value is true (not zero), text will be drawn with a strike-through line.
- If the parameter is omitted, the bold setting is toggled.

You may use the optional Paint pointer parameter <paint_nexp> to specify a Paint object to modify. Normally this parameter is omitted. See **Gr.color**, *Advanced usage* for more information.

Gr.text.underline {{<lexp>}}{,<paint_nexp>}}

Turns underlining on or off on text objects drawn after this command is issued:

- If the value of the underline parameter <lexp> is false (0), text underlining is turned off.
- If the parameter value is true (not zero), drawn text will be underlined.
- If the parameter is omitted, the underline setting is toggled.

You may use the optional Paint pointer parameter <paint_nexp> to specify a Paint object to modify. Normally this parameter is omitted. See **Gr.color**, *Advanced usage* for more information.

Gr.text.setfont {{<font_ptr_nexp>|<font_family_sexp>}} {,<style_sexp>}{,<paint_nexp>}}

Set the text font, specifying typeface and style. Both of these parameters are optional. This command is similar to the older **Gr.text.typeface**, but it is more flexible.

If the font parameter is a numerical expression <font_ptr_nexp>, it must be a font pointer value returned by the **Font.load** command. You cannot modify the style of a font once it is loaded, so the the style parameter <style_sexp> is ignored.

If the font parameter is a string expression <font_family_sexp>, it must specify one of the font families available on your Android device. If your device does not recognize the string, the font is set to the system default font typeface. On most systems, the default is "**sans serif**".

The standard font families are "**monospace**", "**serif**", and "**sans serif**". Some more recent versions of Android also support "**sans-serif**", "**sans-serif-light**", "**sans-serif-condensed**", and "**sans-serif-thin**". The font family names are not case-sensitive: "Serif" or "SERIF" works as well as "serif".

If you omit the font parameter, the command sets the most recently loaded font (see **Font.load**). If you have deleted fonts (see **Font.delete**), the command sets the most recently loaded font that has not been deleted. If you have not loaded any fonts, or if you have cleared them (see **Font.clear**), the command sets the default font family.

If you specify a font family, you can use the style parameter <style_sexp> to change the font's appearance. The parameter value must be one of the style strings shown in the table below. You may use either the full style name or an abbreviation as shown. The parameter is not case-sensitive: "BOLD", "bold", "Bold", and "bOLD" are all the same. If you use any other string, or if you omit the style parameter, the style is set to "**NORMAL**".

Style Name	Abbreviation
"NORMAL"	"N"
"BOLD"	"B"
"ITALIC"	"I"
"BOLD_ITALIC"	"BI"

Notes: The "**monospace**" font family always displays as "**normal**", regardless of the style parameter. Some devices do not support all of the styles.

You may use the optional Paint pointer parameter <paint_nexp> to specify a Paint object to modify. Normally this parameter is omitted. See **Gr.color**, *Advanced usage* for more information.

Gr.text.typeface {{<font_nexp>} {, <style_nexp>} {<paint_nexp>}}

Set the text typeface and style. Both of these parameters are optional. The default value if you omit either parameter is 1. All valid values and their meanings are shown in this table:

The values for <font_nexp> are:		The values for <style_nexp> are:	
1	Default font	1	Normal (not bold or italic)
2	Monospace font	2	Bold
3	Sans-serif font	3	Italic
4	Serif font	4	Bold and italic

This command is similar to the newer **Gr.text.setfont**, except that it is limited to the four typefaces listed in the table. It cannot specify fonts loaded by the **Font.load** command.

Notes: The "Monospace" font (font 2) always displays as "Normal" (style 1), regardless of the style parameter. Some devices do not support all of the styles.

You may use the optional Paint pointer parameter <paint_nexp> to specify a Paint object to modify. Normally this parameter is omitted. See **Gr.color**, *Advanced usage* for more information.

Gr.text.height {<height_nvar>} {, <up_nvar>} {, <down_nvar>}

Returns height information for the current font and text size. All of the parameters are optional; use commas to indicate omitted parameters (see Optional Parameters).

If the **height** variable <height_nvar> is present, it is set to the height in pixels of the space that will be used to print most text in most languages. This is the value you set with **Gr.text.size**. In typeface terminology, it is the "ascent" plus the "descent". The space contains the ascenders of letters like "h" and the descenders of letters like "y".

Some letters, such as the Polish letter "Ż", may not fit in this space. The position of a line high enough to contain all possible characters is returned in the **up** variable <up_nvar>, if it is present.

If the **down** variable <down_nvar> is present, it is set to the "descent" value. This position is low enough to contain the lowest part of all possible characters, such as the tail of a "y".

Gr.text.draw positions text so the the body of the characters sit on a line called the baseline. The **down** and **up** values are reported as offsets from this baseline. The **up** value is negative, because it defines a position above the baseline. The **down** value is positive, because its position is below the baseline. The **height** value is not an offset, so it is always positive. **down - up** is always larger than **height**.

Sometimes you want to know the real screen positions of the top and bottom of the area where your text will be drawn, independent of the actual text you will draw there. The bottom of this area is the **y** coordinate of **Gr.text.draw** plus the **down** value of **Gr.text.height**. For most applications, the top of the text area is the bottom position minus the **height** value of **Gr.text.height**, so **y + down - height**. For some applications (such as a Polish text field), you may need the extra height you get with **y + up**.

```
GR.TEXT.SIZE 40
GR.TEXT.HEIGHT ht, up, dn    % ht is 40
GR.TEXT.DRAW t, x, y, "Hello, World!"
txtBottom = y + dn
txtTop = txtBottom - ht      % good for most applications
txtTop = y + up              % high enough for all possible text (up is negative)
```

Gr.text.width <nvar>, <exp>

Returns the pixel width of a string (from <exp>) in the variable <nvar>.

- If the parameter <exp> is a string expression, the return value is the width of the string as if it were displayed on the screen using the latest text attribute settings: the typeface, size, and style as set by the **Gr.text.*** commands (or default values if you did not set them).
- If the parameter <exp> is a numeric expression, its value must be a text object number from **Gr.text.draw**, or you will get a run-time error. The return value is the width of the text of the object as it would be displayed on the screen by **Gr.render**.

Advanced usage: To calculate dimensions, both **Gr.text.width** and **Gr.get.textbounds** (below) use a text string and a set of text attributes. The text attributes are kept in a Paint object. The source of the string and the Paint depends on the type of the <exp> parameter:

If <exp> is a	value of <exp> is	source of text string is	Source of text attributes is
string expression	the string to measure	value of <exp>	Current Paint (most recent Gr.text.* settings)
numeric expression	a text object number from Gr.text.draw	string from Gr.text.draw (kept in text object)	Paint attached to the text object (see Note , below)

Note: **Gr.text.draw** attaches a Paint to the text object using the text attributes that are current at that time. This is the Paint **Gr.render** uses to display the text on the screen. If you modify this Paint, the changes are reflected in values returned by **Gr.text.width** and **Gr.get.textbounds**, and also shown on the screen with the next **Gr.render**.

Gr.get.textbounds <exp>, left, top, right, bottom

Gets the boundary rectangle of a string as it would be drawn on the screen. The returned coordinate values give you the dimensions of the bounding rectangle but not its location.

The parameter <exp> follows the same rules as **Gr.text.width** (see above) to get a text string and the text attributes (typeface, size, and style) used to measure the string.

The coordinates of the rectangle are reported as if **Gr.text.draw** positioned your text at **0,0**. You get the actual boundaries of a text object by adding the textbounds offsets to the actual **x,y** coordinates of the **Gr.text.draw** command.

In typeface terminology, the coordinate values are offsets from the beginning of the baseline of the text (see **Gr.text.draw** and **Gr.text.height** for more explanation of the lines that define where text is drawn). This is why the value returned for "top" is always a negative number.

If this is confusing, try running this example:

```
GR.OPEN , , , , -1
GR.COLOR 255,255,0,0,0
GR.TEXT.SIZE 40
GR.TEXT.ALIGN 1
s$ = "This is only a test"
GR.GET.TEXTBOUNDS s$,l,t,r,b
PRINT l,t,r,b
x=10 : y=50
GR.RECT rct,l+x,t+y,r+x,b+y
GR.TEXT.DRAW txt,x,y,s$
GR.RENDER
PAUSE 5000
```

Gr.text.draw <object_number_nvar>, <x_nexp>, <y_nexp>, <text_object_sexp>

Creates and inserts a text object (<text_object_sexp>) into the Display List. The text object will use the latest color and text preparatory commands. The <object_number_nvar> returns the Display List object number for this text. This object will not be visible until the **Gr.render** command is called.

Gr.text.draw positions the text so the the bodies of the characters sit on a line called the baseline. The tails of letters like "y" hang below the baseline. The **y** value <y_nexp> sets the location of this baseline.

The **Gr.text.height** command tells you the locations of the various lines used to draw text. The **Gr.text.width** command tells you width of the space in which a specific string will be drawn. The **Gr.get.textbounds** command tells you the locations of the left-, top-, right-, and bottom-most pixels actually drawn for a specific text string.

The **Gr.modify** parameters for **Gr.text.draw** are "x", "y", and "text". The value for "text" is a string representing the new text.

Bitmap Commands

Overview

When a bitmap is created, it is added to a list of bitmaps. Commands that create bitmaps return a pointer to the bitmap. The pointer is an index into the bitmap list. Your program works with the bitmap through the bitmap pointer.

If you want to draw the bitmap on the screen, you must add a graphical object to the Object List. The **Gr.bitmap.draw** command creates a graphical object that holds a pointer to the bitmap. Do not confuse the bitmap with the graphical object. You cannot use the Object Number to access the bitmap, and you cannot use the bitmap pointer to modify the graphical object.

Android devices limit the amount of memory available to your program. Bitmaps may use large blocks of memory, and so may exceed the application memory limit. If a command that creates a bitmap exceeds the limit, the bitmap is not created, and the command returns -1, an invalid bitmap pointer. Your program should test the bitmap pointer to find out if the bitmap was created. If the bitmap pointer is -1, you can call the **GETERROR\$()** function to get information about the error.

If a command exceeds the memory limit, but BASIC! does not catch the out-of-memory condition, your program terminates with an error message displayed on the Console screen. If you return the Editor, a line will be highlighted near the one that exceeded the memory limit. It may not be exactly the right line.

Bitmaps use four bytes of memory for each pixel. The amount of memory used depends only on the width and height of the bitmap. The bitmap is not compressed. When you load a bitmap from a file, the file is usually in a compressed format, so the bitmap will usually be larger than the file.

Gr.bitmap.create <bitmap_ptr_nvar>, width, height

Creates an empty bitmap of the specified width and height. The specified width and height may be greater than the size of the screen, if needed.

Returns a pointer to the created bitmap in the <bitmap_ptr_nvar> variable for use with the other **Gr.bitmap** commands. If there is not enough memory available to create the bitmap, the returned bitmap pointer is -1. Call **GETERROR\$()** for information about the failure.

Gr.bitmap.load <bitmap_ptr_nvar>, <file_name_sexp>

Creates a bitmap from the file specified in the file_name string expression. Returns a pointer to the created bitmap for use with other **Gr.bitmap** commands. If no bitmap is created, the returned bitmap pointer is -1. Call **GETERROR\$()** for information about the failure. Some of the possible causes are:

- The file or resource does not exist.
- There is not enough memory available to create the bitmap.

Bitmap image files are assumed to be located in the "<pref base drive>/rfo-basic/data/" directory.

Note: You may include path fields in the file name. For example, ".././Cougar.jpg" would cause BASIC! to look for Cougar.jpg in the top level directory of the base drive, usually the SD card. "images/Kitty.png"

would cause BASIC! to look in the images(d) sub-directory of the "/sdcard/rfo-basic/data/" ("/sdcard/rfo-basic/data/images/Kitty.png").

Note: Bitmaps loaded with this command cannot be changed with the **Gr.bitmap.drawinto** command. To draw into an image loaded from a file, first create an empty bitmap then draw the loaded bitmap into the empty bitmap.

Gr.bitmap.size <bitmap_ptr_nexp>, width, height

Return the pixel width and height of the bitmap pointed to by <bitmap_ptr_nexp> into the width and height variables.

Gr.bitmap.scale <new_bitmap_ptr_nvar>, <bitmap_ptr_nexp>, width, height {<smoothing_lexp>}

Scales a previously loaded bitmap (<bitmap_ptr_nexp>) to the specified width and height and creates a new bitmap <new_bitmap_ptr_nvar>. The old bitmap still exists; it is not deleted. If there is not enough memory available to create the new bitmap, the returned bitmap pointer is -1. Call **GETERROR\$()** for information about the failure.

Negative values for width and height will cause the image to be flipped left to right or upside down.

Neither the width value nor the height value may be zero.

Use the optional smoothing logical expression (<smoothing_lexp>) to request that the scaled image not be smoothed. If the expression is false (zero) then the image will not be smoothed. If the optional parameter is true (not zero) or not specified then the image will be smoothed.

Gr.bitmap.delete <bitmap_ptr_nexp>

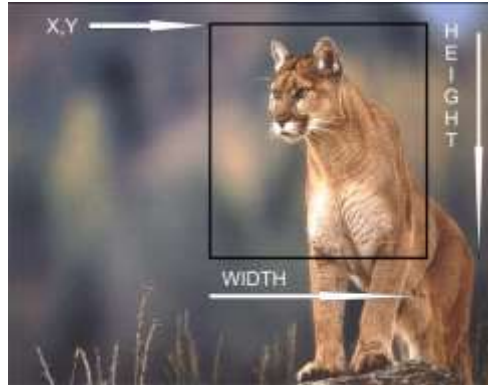
Deletes an existing bitmap. The bitmap's memory is returned to the system.

This does not destroy any graphical object that points to the bitmap. If you do not **Gr.hide** such objects, or remove them from the Display List, you will get a run-time error from the next **Gr.render** command.

Gr.bitmap.crop <new_bitmap_ptr_nvar>, <source_bitmap_ptr_nexp>, <x_nexp>, <y_nexp>, <width_nexp>, <height_nexp>

Creates a cropped copy of an existing source bitmap specified by <source_bitmap_ptr_nexp>. The source bitmap is unaffected; a rectangular section is copied into a new bitmap. A pointer to the new bitmap is returned in <new_bitmap_ptr_nvar>. If there is not enough memory available to create the new bitmap, the returned bitmap pointer is -1. Call **GETERROR\$()** for information about the failure.

The <x_nexp>, <y_nexp> pair specifies the point within the source bitmap that the crop is to start at. The <width_nexp>, <height_nexp> pair defines the size of the rectangular region to crop.



Gr.bitmap.save <bitmap_ptr_nvar>, <filename_sexp>{, <quality_nexp>}

Saves the specified bitmap to a file. The default path is "<pref base drive>/rfo-basic/data/".

The file will be saved as a JPEG file if the filename ends in ".jpg".

The file will be saved as a PNG file if the filename ends in anything else (including ".png").

Gr.bitmap.draw <object_ptr_nvar>, <bitmap_ptr_nexp>, x, y

Creates a graphical object that contains a bitmap and inserts the object into the Object List. The bitmap is specified by the bitmap pointer <bitmap_ptr_nexp>. The bitmap will be drawn with its upper left corner at the provided (x,y) coordinates. The command returns the Object List object number of the graphical object in the <object_ptr_nvar> variable. This object will not be visible until the **Gr.render** command is called.

The alpha value of the latest **Gr.color** will determine the transparency of the bitmap.

The **Gr.modify** parameters for **Gr.bitmap.draw** are "bitmap", "x" and "y".

Gr.get.bmpixel <bitmap_ptr_nvar>, x, y, alpha, red, green, blue

Return the color data for the pixel of the specified bitmap at the specified x, y coordinate. The x and y values must not exceed the length or width of the bitmap.

Gr.bitmap.fill <bitmap_ptr_nexp>, <x_nexp>, <y_nexp>

Change all of the points in an area of a bitmap to the current drawing color. The bitmap pointer parameter <bitmap_ptr_nexp> must specify an existing bitmap. The x and y parameters <x_nexp> and <y_nexp> must specify a point (x,y) in the bitmap. The area to color is a set of connected pixels all the same color. The area may be any shape, and the point (x,y) may be any point in the area.

This command reads actual bitmap pixel colors, so it is affected by the antialiasing setting. If antialiasing is on, the pixels at the edge of the colored area may not be re-colored correctly.

Gr.bitmap.drawinto.start <bitmap_ptr_nexp>

Put BASIC! into the draw-into-bitmap mode.

All draw commands issued while in this mode draw directly into the bitmap. The objects drawn in this mode are not placed into the display list. The object number returned by the draw commands while in this mode is invalid and should not be used for any purpose including **Gr.modify**.

Note: Bitmaps loaded with the **Gr.bitmap.load** command cannot be changed with **Gr.bitmap.drawinto**. To draw into an image loaded from a file, first create an empty bitmap then draw the loaded bitmap into the empty bitmap.

Gr.bitmap.drawinto.end

End the draw-into-bitmap mode. Subsequent draw commands will place the objects into the display list for rendering on the screen. If you wish to display the drawn-into bitmap on the screen, issue a **Bitmap.draw** command for that bitmap.

Paint Commands

Gr.paint.copy {{<src_nexp>}, <dst_nexp>}}

Copy the Paint object at the source pointer <src_nexp> to the destination pointer <dst_next>.

Both parameters are optional. If you wish to specify a destination, you must include a comma, whether or not you specify a source. If either parameter is omitted, or if its value is -1, the current Paint is used.

The Paint already at the destination pointer is replaced. If the destination is the current Paint, a newly-created paint becomes the current Paint.

This command has four forms, depending on which parameters are present:

GR.PAINT.COPY	% Duplicate the current Paint
GR.PAINT.COPY m	% Copy Paint m to the current Paint
GR.PAINT.COPY , n	% Overwrite Paint n so it is the same as the current Paint
GR.PAINT.COPY m, n	% Overwrite Paint n so it is the same as Paint m

Gr.paint.get <object_ptr_nvar>

Gets a pointer (<object_ptr_nvar>) to the last created Paint object. For information about Paint objects, see the section Graphics → Introduction → Paints, above.

This pointer can be used to change the Paint object associated with a draw object by means of the **Gr.modify** command. The **Gr.modify** parameter is "paint".

If you want to modify any of the paint characteristics of an object then you will need to create a current Paint object with those parameters changed. For example:

```
GR.COLOR 255,0,255,0,0
GR.TEXT.SIZE 20
GR.TEXT.ALIGN 2
GR.PAINT.GET the_paint
GR.MODIFY shot, "paint", the_paint
```

changes the current text size and alignment as well as the color.

Gr.paint.reset {<nexp>}

Force the specified Paint to default settings:

Color opaque black (255, 0, 0, 0)

Antialias ON

Style FILL (0)

Minimum stroke width (0.0)

The parameter is optional. If the parameter is omitted or set to -1, a new current Paint is created with default settings.

Rotate Commands

These commands put graphics objects on the Display List like the GR drawing commands, but they don't draw anything. Instead they work as markers in the list. When the renderer sees the start marker, it temporarily rotates its coordinate system. The end marker tells the renderer to restore the coordinate system to normal.

The effect is to rotate or move any objects drawn after **Gr.rotate.start** and before **Gr.rotate.end**.

As with any graphics object, the rotate parameters may be changed with **Gr.modify**. At the next **Gr.render**, the rotated objects will be redrawn in their new positions.

Gr.rotate.start angle, x, y{<obj_nvar>}

Any objects drawn between the **Gr.rotate.start** and **Gr.rotate.end** will be rotated at the specified angle, in degrees, around the specified (x,y) point. If the angle is positive, objects are rotated clockwise.

The optional <obj_nvar> will contain the Object list object number of the **Gr.rotate.start** object. If you are going to use rotated objects in the array for **Gr.NewDI** then you will need to include the **Gr.rotate.start** and **Gr.rotate.end** objects.

The **Gr.modify** parameters for **Gr.rotate.start** are "angle", "x" and "y".

Gr.rotate.start must be eventually followed by **Gr.rotate.end** or you will not get the expected results.

Gr.rotate.end {<obj_nvar>}

Ends the rotated drawing of objects. Objects created after this command will not be rotated.

The optional <obj_nvar> will contain the Object list object number of the **Gr.rotate.end** object. If you are going to use rotated objects in the array for **Gr.NewDI** then you will need to include the **Gr.rotate.start** and **Gr.rotate.end** objects.

Camera Commands

There are three ways to use the camera from BASIC!:

- 1) The device's built in Camera User Interface can be used to capture an image. This method provides access to all the image-capture features that you get when you execute the device's

Camera application. The difference is the image bitmap is returned to BASIC! for manipulation by BASIC! The **Gr.camera.shoot** command implements this mode.

- 2) A picture can be taken automatically when the command is executed. This mode allows for untended, time-sequenced image capture. The command provides for the setting the flash to on, off and auto. The **Gr.camera.autoshoot** command implements this mode.
- 3) The third mode is the **Gr.camera.manualshoot** command which is much like the autoshoot mode. The difference is that a live preview is provided and the image is not captured until the screen is touched.

All pictures are taken at full camera resolution and stored with 100% jpg quality as "<pref base drive>/rfo-basic/data/image.png".

All of these commands also return pointers to bitmaps. The bitmaps produced are scaled down by a factor of 4. You may end up generating several other bitmaps from these returned bitmaps. For example, you may need to scale the returned bitmap to get it to fit onto your screen. Any bitmaps that you are not going to draw and render should be deleted using **Gr.bitmap.delete** to avoid out-of-memory situations.

The Sample Program, f33_camera.bas, demonstrates all the modes of camera operations. It also provides examples of scaling the returned image to fit the screen, writing text on the image and deleting obsolete bitmaps.

The Sample Program, f34_remote_camera.bas, demonstrates remote image capture using two different Android devices.

Gr.camera.select 1|2

Selects the Back (1) or Front(2) camera in devices with two cameras. The default camera is the back (opposite the screen) camera.

If only one camera exists, then the default will be that camera. For example, if the device (such as the Nexus 7) only has a Front Camera then it will be the default camera. If the device does not have any installed camera apps, then there will be a run-time error message, "This device does not have a camera." In addition, a run-time error message will be shown if the device does not have the type of camera (front or back) selected.

Gr.camera.shoot <bm_ptr_nvar>

The command calls the device's built in camera user interface to take a picture. The image is returned to BASIC! as a bitmap pointed to by the bm_ptr numeric variable. If the camera interface does not, for some reason, take a picture, bm_ptr will be returned with a zero value.

Many of the device camera interfaces will also store the captured images somewhere else in memory with a date coded filename. These images can be found with the gallery application. BASIC! is not able to prevent these extraneous files from being created.

Note: Some devices like the Nexus 7 do not come with a built in camera interface. If you have installed an aftermarket camera application then it will be called when executing this command. You can take pictures with the Nexus 7 (or similar devices) using the other commands even if you do not have camera application installed. If the device does not have any installed camera apps, then there will be a run-time error message, "This device does not have a camera."

Gr.camera.autoshoot <bm_ptr_nvar>{, <flash_mode_nexp> {, focus_mode_nexp} }

An image is captured as soon as the command is executed. No user interaction is required. This command can be used for untended, time-sequence image captures.

The optional flash_mode numeric expression specifies the flash operation:

0	Auto Flash
1	Flash On
2	Flash Off
3	Torch
4	Red-eye

The default, if no parameter is given, is Auto Flash.

The optional focus_mode numeric expression specifies the camera focus:

0	Auto Focus
1	Fixed Focus
2	Focus at Infinity
3	Macro Focus (close-up)

The default, if no parameter is given, is Auto Focus.

If you want to specify a focus mode, you must also specify a flash mode.

The command also stores the captured image into the file, "<pref base drive>/rfo-basic/data/image.png".

Gr.camera.manualShoot <bm_ptr_nvar>{, <flash_mode_nexp> {, focus_mode_nexp} }

This command is much like **Gr.camera.autoshoot** except that a live preview is shown on the screen. The image will not be captured until the user taps the screen.

Other Graphics Commands

Gr.screen.to_bitmap <bm_ptr_nvar>

The current contents of the screen will be placed into a bitmap. The pointer to the bitmap will be returned in the bm_ptr variable. If there is not enough memory available to create the bitmap, the returned bitmap pointer is -1. Call **GETERROR\$()** for information about the failure.

Please note the idiosyncratic underscore in the command.

Gr.get.pixel x, y, alpha, red, green, blue

Returns the color data for the screen pixel at the specified x, y coordinate. The x and y values must not exceed the width and height of the screen and must not be less than zero.

To get a pixel from the screen, BASIC! must first create a bitmap from the screen. If there is not enough memory available to create the bitmap, you will get an "out-of-memory" run-time error.

Gr.save <filename_sexp> {<quality_nexp>}

Saves the current screen to a file. The default path is "<pref base drive>/rfo-basic/data/".

The file will be saved as a JPEG file if the filename ends in ".jpg".

The file will be saved as a PNG file if the filename ends in anything else (including ".png").

The optional <quality_nexp> is used to specify the quality of a saved JPEG file. The value may range from 0 (bad) to 100 (very good). The default value is 50. The quality parameter has no effect on PNG files which are always saved at the highest quality level.

Note: The size of the JPEG file depends on the quality. Lower quality values produce smaller files.

Gr.get.type <object_ptr_nexp>, <type_svar>

Get the type of the specified display list object. The type is a string that matches the name of the command that created the object: "point", "circle", "rect", etc. For a complete list of types, see the table in **Gr.Modify**.

If the <object_ptr_nexp> parameter does not specify a valid display list object, the returned type is the empty string, "". You can call the **GETERROR\$()** function to get information about the error.

Gr.get.params <object_ptr_nexp>, <param_array\$[]>

Get the modifiable parameters of the specified display list object. The parameter strings are returned in the <param_array\$[]> in no particular order. The array is specified without an index. If the array exists, it is overwritten. Otherwise a new array is created. The result is always a one-dimensional array.

For a complete list of parameters, see the table in **Gr.Modify**.

Gr.get.position <object_ptr_nexp>, x, y

Get the current x,y position of the specified display list object. If the object was specified with rectangle parameters (left, top, right, bottom) then left is returned in x and top is returned in y. For Line objects, the x1 and y1 parameters are returned.

Gr.move <object_ptr_nexp> {{ dx}{ dy}}

Moves the graphics object by the amounts dx and dy. If the object is a group, all of the graphical objects in the group are moved. The dx and dy parameters are optional. If omitted they default to 0.

Gr.get.value <object_ptr_nexp> {, <tag_sexp>, <value_nvar | value_svar>}...

The value of the parameter named <tag_sexp> ("left", "radius", etc.) in the Display List object <object_ptr_nvar> is returned in the variable <value_nvar> or <value_svar>. This command can return values from only one object at a time, but you may list as many tag/variable pairs as you want.

Most parameters are numeric. Only the **Gr.text.draw** "text" parameter is returned in a string var. The parameters for each object are given with descriptions of the commands in this manual. For a complete list of parameters, see the table in **Gr.Modify**.

Gr.modify <object_ptr_nexp> {, <tag_sexp>, <value_nexp | value_sexp>}...

The value of the parameter named <tag_sexp> in the Display List object <object_ptr_nvar> is changed to the value of the expression <value_nexp> or <value_sexp>. This command can change only one object at a time, but you may list as many tag/value pairs as you want.

With this command, you can change any of the parameters of any object in the Display List. The parameters you can change are given with the descriptions of the commands in this manual. In addition there are two general purpose parameters, "paint" and "alpha" (see below for details). You must provide parameter names that are valid for the specified object.

The results of **Gr.modify** commands will not be observed until a **Gr.render** command executes.

	TYPE	POSITION 1 (numeric)		POSITION 2 (numeric)		ANGLE/RADIUS (numeric)	UNIQUE (various)	PAINT (list ptr)	ALPHA (num)
SHAPES and OBJECTS	arc	left	top	right	bottom	start_angle sweep_angle	fill_mode	paint	alpha
	bitmap	x	y				bitmap	paint	alpha
	circle	x	y			radius		paint	alpha
	line	x1	y1	x2	y2			paint	alpha
	oval	left	top	right	bottom			paint	alpha
	pixels	x	y					paint	alpha
	point	x	y					paint	alpha
	poly	x	y				list	paint	alpha
	rect	left	top	right	bottom			paint	alpha
MODIFIERS	text	x	y				text	paint	alpha
	clip	left	top	right	bottom		RO	paint	alpha
	group						list	paint	alpha
	rotate	x	y			angle		paint	alpha

TABLE NOTES:

- The **TYPE** column shows the string returned by **Gr.get.type** for each graphical object type.
- **Gr.get.position** returns the values in the **POSITION 1** columns.
- All table entries are **Gr.modify** tags (strings). Values of all the tags are numeric except for **"text"**.
- The values of tags in the **UNIQUE** column are either strings (**"text"**) or numbers with special interpretations. **"fill_mode"** is a logical value. **"list"** is a pointer to a list of point coordinates. **"RO"** is a Region Operator as explained in **Gr.clip**.
- **"alpha"** is an integer value from 0 to 256, with 256 interpreted specially. See **General Purpose Parameters**, below.
- You can modify the **Gr.set.pixels** point-coordinates array directly. There is no **Gr.modify** tag.

For example, suppose a bitmap object was created with **Gr.bitmap.draw BM_ptr, galaxy_ptr, 400, 120**.

Executing **gr.modify BM_ptr, "x", 420** would move the bitmap from x = 400 to x = 420.

Executing **gr.modify BM_ptr, "y", 200** would move the bitmap from y = 120 to y = 200.

Executing **gr.modify BM_ptr, "x", 420, "y", 200** would change both x and y at the same time.

Executing **gr.modify BM_ptr, "bitmap", Saturn_ptr** would change the bitmap of an image of a (preloaded) Galaxy to the image of a (preloaded) Saturn.

General Purpose Parameters

When you create a graphical object, all the graphics settings (color, stroke, text settings, and so forth) are captured in a Paint object. You can use the "paint" parameter to replace the Paint object, changing any graphics setting you want to. See the **Gr.paint.get** command description (below) for more details.

Normally, graphical objects get their alpha channel value (transparency) from the latest **Gr.color** command. You can change the "alpha" parameter to any value from 0 to 255. Setting alpha to 256 tells BASIC! to use the alpha from the latest color value.

For example, you can make an object slowly appear and disappear, just by changing its alpha with **Gr.modify**.

```
Do
    For a = 1 to 255 step 10
        gr.modify object,"alpha",a
        gr.render
        pause 250
    next a

    For a = 255 to 1 step -10
        gr.modify object,"alpha",a
        gr.render
        pause 250
    next a
until 0
```

GR_COLLISION(<object_1_nexp>, <object_2_nexp>)

GR_COLLISION() is a function, not a command. The variables <object_1_nvar> and <object_2_nvar> are the object pointers returned when the objects were created.

If the boundary boxes of the two objects overlap then the function will return true (not zero). If they do not overlap then the function will return false (zero).

Objects that may be tested for collision are: rectangle, bitmap, circle, arc, oval, and text. In the case of a circle, an arc, an oval, or text, the object's rectangular boundary box is used for collision testing, not the actual drawn object.

Gr.clip <object_ptr_nexp>, <left_nexp>, <top_nexp>, <right_nexp>, <bottom_nexp>{<RO_nexp>}

Objects that are drawn after this command is issued will be drawn only within the bounds (clipped) of the clip rectangle specified by the "left, top, right, bottom" numeric expressions.

The final parameter is the Region Operator, <RO_nexp>. The Region Operator prescribes how this clip will interact with everything else you are drawing on the screen or bitmap. If you issue more than one **Gr.clip** command, the RO prescribes the interaction between the current **Gr.clip** rectangle and the previous one. The RO values are:

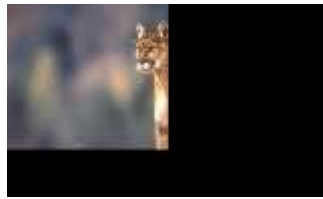
0	Intersect
1	Difference
2	Replace
3	Reverse Difference
4	Union
5	XOR

The Region Operator parameter is optional. If it is omitted, the default action is **Intersect**.

Examples:



Original

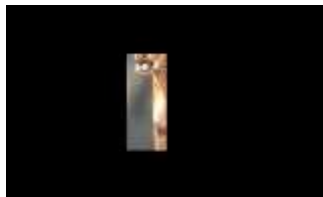


Clip 1

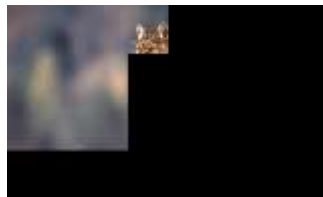


Clip 2

Clip 2 applied to Clip 1 with RO parameter on Clip 2



0 = Intersect



1 = Difference



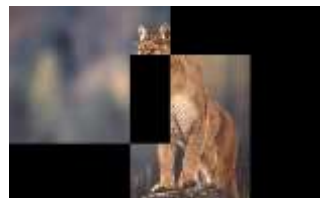
2 = Replace



3 = Reverse Difference



4 = Union



5 = XOR

Gr.clip is a display list object. It can be modified with **Gr.modify**. The modify parameters are "left", "top", "right", "bottom", and "RO".

The **Gr.show** and **Gr.hide** commands can be used with the **Gr.clip** object.

Gr.newDL <dl_array[{<start>,<length>}]>

Replaces the existing display list with a new display list read from a numeric array (dl_array[]) or array segment (dl_array[start,length]) of object numbers. Zero values in the array will be treated as null objects in the display list. Null objects will not be drawn nor will they cause run-time errors.

See the Display List subtopic in this chapter for a complete explanation.

See the Sample Program file, f24_newdl, for a working example of this command.

Gr.getDL <dl_array[]> {, <keep_all_objects_lexp> }

Writes the current Display List into the numeric array <dl_array[]>. The array is specified without an index. If the array exists, it is overwritten. Otherwise a new array is created. The result is always a one-dimensional array. If the Display List is empty, the array will have one entry that does not display anything.

By default, objects hidden with **Gr.hide** are not included in the returned array. To get all objects, including hidden objects, set the optional `keep_all_objects` flag to true (any non-zero value).

Audio Interface

Introduction

The Audio Interface

BASIC! uses the Android Media Player interface for playing music files. This interface is not the most stable part of Android. It sometimes gets confused about what it is doing. This can lead to random "Forced Close" events. While these events are rare, they do occur.

Audio File Types

The file types you can play depend on your device and the version of Android it runs. For a current list check the [Android documentation](#). Here is a partial summary:

Audio File Type	Played by Android Version
AAC AMR MIDI MP3 OGG WAV WMA	all
FLAC	3.1+
AAC-ELD	4.1+
MKV	5.0+

Commands

Audio files must be loaded into the Audio File Table (AFT) before they can be played. Each audio file in the AFT has a unique index which is returned by the `audio.load` command.

Audio.load <aft_nvar>, <filename_sexp>

Loads a music file or internet stream into the Audio File Table. The AFT index is returned in <aft_nvar>. If the file or stream can't be loaded, the <aft_nvar> is set to 0. Your program should test the AFT index to find out if the audio was loaded. If the AFT index is 0, you can call the **GETERROR\$()** function to get information about the error. If you use index 0 in another **Audio** command you will get a run-time error.

To load a music file, specify an optional path and a filename. For example:

"Blue Danube Waltz.mp3" would access "<pref base drive>/rfo-basic/data/Blue Danube Waltz.mp3".

"../Music/Blue Danube Waltz.mp3" would access "<pref base drive>/Music/Blue Danube Waltz.mp3".

To load an internet stream, specify a full URL.

Audio.play <aft_nexp>

Selects the file from the Audio File Table pointed to by <aft_nexp> and begins to play it. There must not be an audio file already playing when this command is executed. If there is a file playing, execute `audio.stop` first.

The music stops playing when the program stops running. To simply start a music file playing and keep it playing, keep the program running. This infinite loop will accomplish that:

```
Audio.load ptr, "my_music.mp3"  
Audio.play ptr  
Do  
  Pause 5000  
Until 0
```

Audio.stop

Audio.stop terminates the currently-playing music file. The command will ignored is no file is playing. It is best to precede each audio.play command with an audio.stop command.

Audio.pause

Pause is like stop except that the next audio.play for this file will resume the play at the point where the play was paused.

Audio.loop

When the currently playing file reaches the end of file, the file will restart playing from the beginning of the file. There must be a currently playing file when this command is executed.

Audio.volume <left_nexp>, <right_nexp>

Changes the volume of the left and right stereo channels. There must be a currently playing file when this command is executed.

The values should range between 0.0 (lowest) to 1.0 (highest). The human ear perceives the level of sound changes on a logarithmic scale. The ear perceives a 10db change as twice as loud. A 20db change would be four times as loud.

A 1 db change would be about 0.89. One way to implement a volume control would be set up a volume table with 1db level changes. The following code creates a 16 step table.

```
dim volume[16]  
x =1  
volume [1] = x  
for i = 2 to 16  
  x = x * 0.89  
  volume [i] = x  
next i
```

Your code can select volume values from the table for use in the audio.volume command. The loudest volume would be volume[1].

Audio.position.current <nvar>

The current position in milliseconds of the currently playing file will be returned in <nvar>.

Audio.position.seek <nexp>

Moves the playing position of the currently playing file to <nexp> expressed in milliseconds.

Audio.length <length_nvar>, <aft_nexp>

Returns the total length of the file in the Audio File Table pointed to by <aft_nexp>. The length in milliseconds will be returned in <length_nvar>.

Audio.release <aft_nexp>

Releases the resources used by the file in the Audio File Table pointed to by <aft_nexp>. The file must not be currently playing. The specified file will no longer be able to be played.

Audio.isdone <lvar>

If the current playing file is still playing then <lvar> will be set to zero otherwise it will be set to one. This can be used to determine when to start playing the next file in a play list.

```
Audio.play f[x]
Do
    Audio.isdone isdone
    Pause 1000
Until isdone
```

Audio.record.start <fn_svar>

Start audio recording using the microphone as the audio source. The recording will be saved to the specified file. The file must have the extension .3GP. Recording will continue until the audio.record.stop command is issued.

Audio.record.stop

Stops the previously started audio recording.

SoundPool

Introduction

A SoundPool is a collection of short sound bites that are preloaded and ready for instantaneous play. SoundPool sound bites can be played while other sounds are playing, either while other sound bites are playing or over a currently playing sound file being played by means of Audio.play. In a game, the Audio.play file would be the background music while the SoundPool sound bites would be the game sounds (Bang, Pow, Screech, etc).

A SoundPool is opened using the SoundPool.open command. After the SoundPool is opened, sound bites will be loaded into memory from files using the SoundPool.load command. Loaded sound bites can be played over and over again using the SoundPool.play command.

A playing sound is called a sound stream. Individual sound streams can be paused (`SoundPool.pause`), individually or as a group, resumed (`SoundPool.resume`) and stopped (`SoundPool.stop`). Other stream parameters (priority, volume and rate) can be changed on the fly.

The `SoundPool.release` command closes the `SoundPool`. A new `SoundPool` can then be opened for a different phase of the game. `SoundPool.release` is automatically called when the program run is terminated.

Commands

Soundpool.open <MaxStreams_nexp>

The `MaxStreams` expression specifies the number of `Soundpool` streams that can be played at once. If the number of streams to be played exceeds this value, the lowest priority streams will be terminated.

Note: A stream playing via `audio.play` is not counted as a `Soundpool` stream.

Soundpool.load <soundID_nvar>, <file_path_sexp>

The file specified in `<file_path_sexp>` is loaded. Its sound ID is returned in `<soundID_nvar>`. The sound ID is used to play the sound and also to unload the sound. The sound ID will be returned as zero if the file was not loaded for some reason.

The default file path is `"sdcard/rfo-basic/data/"`

Note: It can take a few hundred milliseconds for the sound to be loaded. Insert a `"Pause 500"` statement after the load if you want to play the sound immediately following the load command.

Soundpool.unload <soundID_nexp>

The specified loaded sound is unloaded.

Soundpool.play <streamID_nvar>, <soundID_nexp>, <rightVolume_nexp>, <leftVolume_nexp>, <priority_nexp>, <loop_nexp>, <rate_nexp>

Starts the specified sound ID playing.

The stream ID is returned in `<streamID_nvar>`. If the stream was not started, the value returned will be zero. The stream ID is used to pause, resume and stop the stream. It is also used in the stream modification commands (`Soundpool.setrate`, `Soundpool.setvolume`, `Soundpool.setpriority` and `Soundpool.setloop`).

The left and right volume values must be in the range of 0 to 0.99 with zero being silent.

The priority is a positive value or zero. The lowest priority is zero.

The loop value of -1 will loop the playing stream forever. Values other than -1 specify the number of times the stream will be replayed. A value of 1 will play the stream twice.

The rate value changes the playback rate of the playing stream. The normal rate is 1. The minimum rate (slow) is 0.5. The maximum rate (fast) is 1.85.

Soundpool.setvolume <streamID_nexp>, <leftVolume_nexp>, <rightVolume_nexp>

Changes the volume of a playing stream.

The left and right volume values must be in the range of 0 to 0.99 with zero being silent.

Soundpool.setrate <streamID_nexp>, <rate_nexp>

Changes the playback rate of the playing stream.

The normal rate is 1. The minimum rate (slow) is 0.5. The maximum rate (fast) is 1.85.

Soundpool.setpriority <streamID_nexp>, <priority_nexp>

Changes the priority of a playing stream.

The lowest priority is zero.

Soundpool.pause <streamID_nexp>

Pauses the playing of the specified stream. If the stream ID is zero, all streams will be paused.

Soundpool.resume <streamID_nexp>

Resumes the playing of the specified stream. If the stream ID is zero, all streams will be resumed.

Soundpool.stop <streamID_nexp>

Stops the playing of the specified stream.

Soundpool.release

Closes the SoundPool and releases all resources. Soundpool.open can be called to open a new SoundPool.

GPS

These commands provide access to the raw location data reported by an Android device's GPS hardware. Before attempting to use these commands, make sure that you have GPS turned on in the Android Settings Application.

The Sample Program file, f15_gps.bas is a running example of the use of the GPS commands.

There are two kinds of data reports: GPS status and location data. They are not reported at the same time, so there is no guarantee that overlapping information matches. For example, the location data report includes a count of the satellites used in the most recent location fix. The same information can be derived from the GPS status report. If number of detected satellites changes between reports, the two numbers do not agree.

GPS Control commands

Gps.open {{<status_nvar>},{<time_nexp>},{<distance_nexp>}}

Connects to the GPS hardware and starts it reporting location information. This command must be issued before using any of the other GPS commands.

The three parameters are all optional; use commas to indicate missing parameters. The parameters are available for advanced usage. The most common way to use this command is simply **GPS.open**.

If you provide a status return variable <status_nvar>, it is set to 1.0 (TRUE) if the open succeeds, or 0.0 (FALSE) if the open fails. If the open fails, you may get information about the failure from the **GETERROR\$()** function.

The time interval expression <time_nexp> sets the minimum time between location updates. The time is in milliseconds. If you do not set an interval, it defaults to the minimum value allowed by your Android device. This is typically one second. Note: to reduce battery usage, Android recommends a minimum interval of five minutes.

If you provide a distance parameter <distance_nexp>, it is a numerical expression that sets the minimum distance between location updates, in meters. That is, your program will not be informed of location changes until your device has moved at least as far as the minimum distance setting. If you do not set a distance, any location change that can be detected will be reported.

This command attempts to get an initial "last known location". If the GPS hardware does not report a last known location, BASIC! tries to get one from the network location service. If neither source can provide one, the location information is left empty. If you use GPS commands to get location information before the GPS hardware starts reporting current location information, you will get this "last known location" data. The last known location may be stale, hours or days old, and so may not be useful.

Gps.close

Disconnects from the GPS hardware and stops the location reports. GPS is automatically closed when you stop your BASIC! program. GPS is not turned off if you tap the HOME key while your GPS program is running.

Gps.status {{<status_var>}, {<infix_nvar>},{inview_nvar}, {<sat_list_nexp>}}

Returns the data from a GPS status report. The parameters are all optional; use commas to indicate omitted parameters (see Optional Parameters).

This kind of report contains the type of the last GPS event and a list of the satellites that were detected by the GPS hardware when that event occurred. As a convenience, this command analyzes the satellite list to report how many satellites were detected ("in view") and how many of those were used in the last location fix ("in fix").

The GPS status report is not timestamped, and the first event reported to your program may be stale. Do not rely on the data from the first status report alone to determine when the GPS hardware gets a current location fix..

<status_var>: If provided, this variable returns the type of last GPS event that occurred. If you provide a numeric variable, the event type is reported as a number. If it is a string variable, the event type is reported as an English-language event name.

Event Number	Event Name	Meaning
1	Started	The GPS system has been started, no location fixed yet
2	Stopped	The GPS system has been stopped
3	First Fix	The GPS system has received its first location fix since starting
4	Updated	The GPS system has updated its location data

<infix_nvar>: If provided, this numeric variable returns the number of satellites used in the last location fix. This is the number of satellites in the satellite list describe below whose "infix" value is TRUE (non-zero). If the status report could not get a satellite list the number is unknown, so the variable is set to -1.

<inview_nvar>: If provided, this numeric variable returns the number of satellites detected by the GPS hardware. This is the number of satellites in the satellite list described below that have current data. It is not necessarily the size of the list. If the status report could not get a satellite list the number is unknown, so the variable is set to -1.

<sat_list_nexp>: If provided, the value of this numeric expression is used as a list pointer. If the value is not a valid numeric list pointer, and the numeric expression is a single numeric variable, then a new list is created and the numeric variable is set to point to the new list.

The satellite list is a list of bundle pointers. When the GPS system reports GPS status, it provides data collected from the satellites it can detect. The data from each satellite is put in a bundle. The satellite list has pointers to all of the satellite data bundles. You can use these pointers with any **Bundle** command, just like any other bundle pointer.

If you provide an existing list, any bundles already in the list are cleared, except for the identifying pseudo-random number (PRN). Anything else in the list is discarded. Then the new satellite data is written into the satellite bundle. This is done so that a satellite that is lost and then regained will be remembered in the satellite bundle, but its stale data will not be kept.

The number of satellite bundles with complete data matches the value of the **<inview_var>**. These bundles are listed first. Any cleared bundles for satellites not currently visible are at the end of the list.

Each satellite bundle has five key/value pairs. All values are numeric. The value of "infix" is interpreted as logical (Boolean).

KEY	VALUE
prn	Pseudo-Random Number assigned to the satellite for identification

elevation	Elevation in degrees
azimuth	Azimuth in degrees
snr	Signal-to-noise ratio: a measure of signal strength
infix	TRUE (non-zero) if the satellite's data was used in the last location fix, else FALSE (0.0)

This is the only GPS command that returns information from both kinds of GPS data. The satellite count returned in <count_nvar> comes from the location data report, and the satellite list returned in the satellite list comes from the GPS status report. If nothing changes between reports, the number of satellites with infix set TRUE is the same as the satellite count value, but this condition cannot be guaranteed.

The satellite count value is also returned by the **GPS.location** command. The satellite list is also returned or updated by the **GPS.status** command. This command, **GPS.satellites**, is retained for backward-compatibility and for convenience.

For example, let's say the most recent GPS status report had data from three satellites with PRNs 4, 7, and 29.

```
GPS.OPEN sts
GPS.STATUS , , inView, sats
DEBUG.DUMP.LIST sats           % may print 7.0, 29.0, 4.0
```

Assume appropriate delays after the **GPS.open** and that **DEBUG** is enabled. Another GPS status report may report data from satellites 4, 7, and 8. Then the list dump might show 7.0, 4.0, 8.0, 29.0. The order is unpredictable, except that 29.0 will be last, because it is not currently visible. In both cases, the value of inView is 3.0.

Debug.dump.bundle of the satellite bundle with PRN 4 might show this:

```
Dumping Bundle 11
prn: 4.0
snr: 17.0
infix: 0.0
elevation: 25.0
azimuth: 312.0
```

GPS Location commands

These commands report the values returned by the most recent GPS location report. The **Gps.satellites** command also returns the list of satellites contained in a GPS status report.

A location report contains:

- the location provider
- the number of satellites used to generate the data in the report
- the time when the data was reported
- an estimate of the accuracy of the location components
- the location components:

- latitude
- longitude
- altitude
- bearing
- speed

There are individual commands available to read each element of a location report. If you use separate GPS commands to read different components of the location data, you don't know if the different components came from the same location report. To be certain of consistent data, get all of the location components from a single **Gps.location** command.

Gps.location {{<time_nvar>}, {<prov_svar>}, {<count_nvar>}, {<acc_nvar>}, {<lat_nvar>}, {<long_nvar>}, {<alt_nvar>}, {<bear_nvar>}, {<speed_nvar>}}

Returns the data from a single GPS location report. It returns all of the data provided by all of the individual GPS location component commands below, except that it does not return the satellite list of the **Gps.satellites** command.

The parameters are all optional; use commas to indicate missing parameters (see Optional Parameters). All of the parameters are variable names, so if any parameter is not provided, the corresponding data is not returned.

The parameters are:

<time_nvar>: time of the location fix, in milliseconds since the epoch, as reported by **Gps.time**.

<prov_svar>: the location provider, as reported by **Gps.provider**.

<count_nvar>: the number of satellites used to generate the location fix, as reported by **Gps.satellites**.

<acc_nvar>: an estimate of the accuracy of the location fix, in meters, as reported by **Gps.accuracy**.

<lat_nvar>: current latitude, in decimal degrees, as reported by **Gps.latitude**.

<long_nvar>: current longitude, in decimal degrees, as reported by **Gps.longitude**.

<alt_nvar>: current altitude, in meters, as reported by **Gps.altitude**.

<bear_nvar>: current bearing, in compass degrees, as reported by **Gps.bearing**.

<speed_nvar>: current speed, in meters per second, as reported by **Gps.speed**.

Gps.time <nvar>

Returns the time of the last GPS fix in milliseconds since "the epoch", January 1, 1970, UTC.

Gps.provider <svar>

Returns the name of the location provider in <svar>. Normally this is "gps". The first time you read location data, you get the last known location, which may come from either the GPS hardware or the network location service. If it came from the network, this command returns "network". If neither provider reported a last known location, the provider <svar> is the empty string, "".

Gps.satellites {{<count_nvar>}, {<sat_list_nexp>}}

Returns the number of satellites used for the last GPS fix and a list of the satellites known to the GPS hardware.

Both parameters are optional. If you omit <count_nvar> but use <sat_list_nexp>, keep the comma.

If you provide a numeric variable <count_nvar>, it is set to the number of satellites used for the most recent location data. If the location report did not provide a satellite count, <count_nvar> is set to -1.

For a description of the satellite list pointer expression <sat_list_nexp>, see the <sat_list_nexp> parameter of the **Gps.status** command, above.

Gps.accuracy <nvar>

Returns the accuracy level in <nvar>. If non-zero, this is an estimate of the uncertainty in the reported location, measured in meters. A value of zero means the location is unknown.

Gps.latitude <nvar>

Returns the latitude in decimal degrees in <nvar>.

Gps.longitude <nvar>

Returns the longitude in decimal degrees in <nvar>.

Gps.altitude <nvar>

Returns the altitude in meters in <nvar>.

Gps.bearing <nvar>

Returns the bearing in compass degrees in <nvar>.

Gps.speed <nvar>

Returns the speed in meters per second in <nvar>.

Sensors

Introduction

Android devices can have several types of Sensors. Currently, Android's pre-defined Sensors are:

Name of Sensor	Type	Notes
Accelerometer	1	As of API 3 (Cupcake)
Magnetic Field	2	As of API 3
Orientation	3	As of API 3, deprecated API 8
Gyroscope	4	As of API 3
Light	5	As of API 3
Pressure	6	As of API 3
Temperature	7	As of API 3, deprecated API 14

Proximity	8	As of API 3
Gravity	9	As of API 9 (Gingerbread)
Linear Acceleration	10	As of API 9
Rotation Vector	11	As of API 9
Relative Humidity	12	As of API 14 (Ice Cream Sandwich)
Ambient Temperature	13	As of API 14
Uncalibrated Magnetic Field	14	As of API 18 (Jellybean MR2)
Game Rotation Vector	15	As of API 18
Uncalibrated Gyroscope	16	As of API 18
Significant Motion	17	As of API 18
Step Detector	18	As of API 19 (KitKat)
Step Counter	19	As of API 19
Geomagnetic Rotation Vector	20	As of API 19

Some details about (most) of these sensors can be found at [Android's Sensor Event](http://developer.android.com/reference/android/hardware/SensorEvent.html) (<http://developer.android.com/reference/android/hardware/SensorEvent.html>) web page.

Not all Android devices have all of these Sensors. Some Android devices may have none of these sensors. The BASIC! command, `sensors.list`, can be used to provide an inventory of the sensors available on a particular device.

Some newer devices may have sensors that are not currently supported by BASIC! Those sensors will be reported as "Unknown, Type = NN" where NN is the sensor type number.

Sensor Commands

`Sensors.list <sensor_array$[]>`

Writes information about the sensors available on the Android device into the `<sensor_array$[]>` parameter. If the array exists, it is overwritten. Otherwise a new array is created. The result is always a one-dimensional array.

The array elements contain the names and types of the available sensors. For example, one element may be "Gyroscope, Type = 4". The following program snippet prints the elements of the sensor list.

```
SENSORS.LIST sensorarray$[]
ARRAY.LENGTH size, sensorarray$[]
FOR index = 1 TO size
    PRINT sensorarray$[ index ]
NEXT index
END
```

`Sensors.open <type_nexp>{:<delay_nexp>}{, <type_nexp>{:<delay_nexp>}, ...}`

Opens a list of sensors for reading. The parameter list is the type numbers of the sensors to be opened, followed optionally by a colon (':') and a number (0, 1, 2, or 3) that specifies the delay in sampling the sensor. 3 is the default (slowest).

This table gives a general idea of what the rate values mean. The delay values are only "suggestions" to the sensors, which may alter the real delays, and do not apply to all sensors. Faster settings use more battery.

Value	Name	Typical Delay	Typical Usage
3	Normal	200 ms	Default: suitable for screen orientation changes
2	UI	60 ms	Rate suitable for the user interface
1	Game	20 ms	Rate suitable for game play
0	Fastest	0 ms	Sample as fast as possible

Example:

```
SENSORS.OPEN 1:1, 3           % Monitor the Acceleration sensor at Game rate
                                % and the Orientation sensor at Normal rate.
```

This command must be executed before issuing any **Sensors.read** commands. You should only open the sensors that you actually want to read. Each sensor opened increases battery drain and the background CPU usage.

BASIC! uses the colon character to separate multiple commands on a single line. The use of colon in this command conflicts with that feature, so you must use caution when using both features together.

If you put any colons on a line after this command, the preprocessor **always** assumes the colons are part of the command and not command separators. The **Sensors.open** command **must** be either on a line by itself or placed last on a multi-command line.

Sensors.read sensor_type_nexp, p1_nvar, p2_nvar, p3_nvar

This command returns that latest values from the sensors specified by the "sensor_type" parameters. The values are returned are placed into the p1, p2 and p3 parameters. The meaning of these parameters depends upon the sensor being read. Not all sensors return all three parameter values. In those cases, the unused parameter values will be set to zero. See [Android's Sensor Event](#) web page for the meaning of these parameters.

Sensors.close

Closes the previously opened sensors. The sensors' hardware will be turned off preventing battery drain. Sensors are automatically closed when the program run is stopped via the BACK key or Menu->Stop.

System

The **System** commands provide for the execution of System commands on non-rooted devices.

The **SU** commands provide for the execution of Superuser commands on rooted devices. See the Sample Program, f36_superuser.bas, for an example using these commands.

The **App** commands provide for sending messages through the Android system to other applications on your Android device.

The **App** commands are not related to **System** or **SU**, but they are typically used to replace commands of the form **System.write "am start <parameter-list>"** or **System.write "am broadcast <parameter-list>"**.

System Commands

System.open

Opens a shell to execute system commands. The working directory is set to "**rfo-basic**". If the working directory does not exist, it is created. If you open a command shell with either **Su.open** or **System.open**, you can't open another one of either type without first closing the open one.

System.write <sexp>

Executes a System command.

System.read.ready <nvar>

Tests for responses from a **System.write** command. If the result is non-zero, then response lines are available.

Not all System commands return a response. If there is no response returned after a few seconds then it should be assumed that there will be no response.

System.read.line <svar>

Places the next available response line into the string variable.

System.close

Exits the System Shell mode.

Superuser Commands

Su.open

Requests Superuser permission. If granted, opens a shell to execute system commands. The working directory is set to /. If you open a command shell with either **Su.open** or **System.open**, you can't open another one of either type without first closing the open one.

Su.write <sexp>

Executes a Superuser command.

Su.read.ready <nvar>

Tests for responses from a **Su.write** command. If the result is non-zero, then response lines are available.

Not all Superuser commands return a response. If there is no response returned after a few seconds then it should be assumed that there will be no response.

Su.read.line <svar>

Places the next available response line into the string variable.

Su.close

Exits the Superuser mode.

App Commands

App.broadcast <action_sexp>, <data_uri_sexp>, <package_sexp>, <component_sexp>, <mime_type_sexp>, <categories_sexp>, <extras_bptr_nexp>, <flags_nexp>

Creates a system message and broadcasts it to other applications on your device. The message is called an Intent. The Intent will be received by any application that has the right Intent Filter.

All of the parameters are optional; use commas to indicate omitted parameters (see Optional Parameters). See **App.start**, below, for parameter definitions.

If there is no app that can receive your broadcast, the broadcast is ignored. You can detect this condition only by calling the **GETERROR\$()** function. If an app is available to receive the broadcast, **GETERROR\$()** returns "No error".

App.start <action_sexp>, <data_uri_sexp>, <package_sexp>, <component_sexp>, <mime_type_sexp>, <categories_sexp>, <extras_bptr_nexp>, <flags_nexp>

Sends a message to the system, called an Intent, requesting a specific application or type of application to start. If more than one app can handle the request, the system puts up a chooser for you.

If there is no app that can handle the request, the Intent is ignored. You can detect this condition only by calling the **GETERROR\$()** function. If an app is available to launch, **GETERROR\$()** returns "No error".

All of the parameters are optional. Use commas to indicate omitted parameters (see Optional Parameters). You will almost never need to use all of the parameters in one command.

The first six parameters are string expressions: action, data URI, package name, component name, MIME type, and a list of categories separated by commas (the commas are part of the string expression).

The last two parameters are numeric expressions. One is a pointer to a bundle that contains "extras" that are attached to the message. The other is a single number representing one or more flag values.

For parameter values, consult the documentation of the Android system and the app you want to start.

Parameter	Meaning	am Command Equivalent
action	An Action defined by Android or by the target application	-a
data URI	Data or path to data; BASIC! URI-encodes this string	-d
package name	Name of the target application's package (sometimes called ID)	-n [Note 2]
component name	Name of a component within the target application	-n [Note 2]
MIME type	MIME-type of the data, may be used without a data URI	-t
categories	Comma-separated list of Intent categories	-c [Note 3]
"extras" bundle ptr	Pointer to an existing bundle containing "extras" values	-e [Note 4]
flags	Sum of numeric values of one or more flags	-f

Notes:

1. You must use string or numeric values, not Android-defined constants, for actions, types, categories, and flags. For example, you can use the string "**android.intent.action.MAIN**", but you can not use the Android symbol **ACTION_MAIN**.
2. If you specify a component name, you must also specify the package name, even though the package name is often part of the component name. For example, these are equivalent:

```
SYSTEM.WRITE "am -n com.android.calculator2.Calculator"  
APP.START , , "com.android.calculator2", "com.android.calculator2.Calculator"
```

Usually you can use this pattern:

```
pkg$ = "com.android.calculator2" : comp$ = pkg$ + ".Calculator"  
APP.START , , pkg$, comp$
```

3. A categories parameter may contain several categories separated by commas, for example:

```
cats$ = "android.intent.category.BROWSABLE, android.intent.category.MONKEY"  
cats$ = cat1$ + "," + cat2$ + "," + cat3$
```
4. You must create and populate the "extras" bundle before passing its pointer to an **App** command. Presently only string extras and float extras are supported (the BASIC! data types).
5. When your program uses an Intent to start another app, the second app can use another Intent to return results. BASIC! does not yet support this return path. Your program can retrieve data that another app leaves in a file or in the clipboard, but it cannot yet retrieve data from a return Intent.

Appendix A – Command List

! - Single Line Comment, 52
!! - Block Comment, 52
- Format Line, 27
% - Middle of Line Comment, 52
? {<exp> {,|;}} ..., 96
ABS(<nexp>), 56
ACOS(<nexp>), 60
App.broadcast <action_sexp>, <data_uri_sexp>, <package_sexp>, <component_sexp>,
 <mime_type_sexp>, <categories_sexp>, <extras_bptr_nexp>, <flags_nexp>, 199
App.start <action_sexp>, <data_uri_sexp>, <package_sexp>, <component_sexp>, <mime_type_sexp>,
 <categories_sexp>, <extras_bptr_nexp>, <flags_nexp>, 199
Array.average <Average_nvar>, Array[{<start>,<length>}], 41
Array.copy SourceArray[{<start>,<length>}], DestinationArray[{<->{<start_or_extras>}], 41
Array.copy SourceArray[{<start>,<length>}], DestinationArray[{<->{<start_or_extras>}], 41
Array.delete Array[], Array\$[] ..., 41
Array.dims Source[{, {Dims[]}{, NumDims}}, 41
Array.fill Array[{<start>,<length>}], <sexp>, 42
Array.fill Array[{<start>,<length>}], <nexp>, 42
Array.length n, Array[{<start>,<length>}], 42
Array.length n, Array[{<start>,<length>}], 42
Array.load Array\$[], <sexp>, ..., 42
Array.load Array[], <nexp>, ..., 42
Array.max <max_nvar> Array[{<start>,<length>}], 42
Array.min <min_nvar>, Array[{<start>,<length>}], 42
Array.reverse Array\$[{<start>,<length>}], 43
Array.reverse Array[{<start>,<length>}], 43
Array.search Array\$[{<start>,<length>}], <value_sexp>, <result_nvar>{,<start_nexp>}, 43
Array.search Array[{<start>,<length>}], <value_nexp>, <result_nvar>{,<start_nexp>}, 43
Array.shuffle Array[{<start>,<length>}], 43
Array.sort Array[{<start>,<length>}], 43
Array.std_dev <sd_nvar>, Array[{<start>,<length>}], 43
Array.sum <sum_nvar>, Array[{<start>,<length>}], 43
Array.variance <v_nvar>, Array[{<start>,<length>}], 43
ASCII(<sexp>{, <index_nexp>}), 61
ASIN(<nexp>), 60
ATAN(<nexp>), 60
ATAN2(<nexp_y>, <nexp_x>), 60
Audio.isdone <lvar>, 188
Audio.length <length_nvar>, <aft_nexp>, 188
Audio.load <aft_nvar>, <filename_sexp>, 186
Audio.loop, 187
Audio.pause, 187
Audio.play <aft_nexp>, 186
Audio.position.current <nvar>, 187
Audio.position.seek <nexp>, 188
Audio.record.start <fn_svar>, 188

Audio.record.stop, 188
 Audio.release <aft_nexp>, 188
 Audio.stop, 187
 Audio.volume <left_nexp>, <right_nexp>, 187
 Back.resume, 90
 BACKGROUND(), 62
 Background.resume, 145
 BAND(<nexp1>, <nexp2>), 56
 BIN\$(<nexp>), 69
 BIN(<sexp>), 61
 BNOT(<nexp>, 56
 BOR(<nexp1>, <nexp2>), 56
 Browse <url_sexp>, 120
 Bt.close, 128
 Bt.connect {0|1}, 128
 Bt.device.name <svar>, 130
 Bt.disconnect, 128
 Bt.onReadReady.resume, 129
 Bt.open {0|1}, 127
 Bt.read.bytes <svar>, 130
 Bt.read.ready <nvar>, 129
 Bt.reconnect, 128
 Bt.set.UUID <sexp>, 130
 Bt.status {{<connect_var>}{, <name_svar>}{, <address_svar>}}, 128
 Bt.write {<exp> {,|;}} ..., 129
 Bundle.clear <pointer_nexp>, 50
 Bundle.contain <pointer_nexp>, <key_sexp>, <contains_nvar>, 50
 Bundle.create <pointer_nvar>, 48
 Bundle.get <pointer_nexp>, <key_sexp>, <nvar>|<svar>, 49
 Bundle.keys <bundle_ptr_nexp>, <list_ptr_nexp>, 49
 Bundle.put <pointer_nexp>, <key_sexp>, <value_nexp>|<value_sexp>, 49
 Bundle.remove <pointer_nexp>, <key_sexp>, 50
 Bundle.type <pointer_nexp>, <key_sexp>, <type_svar>, 50
 BXOR(<nexp1>, <nexp2>), 56
 Byte.close <file_table_nexp>, 112
 Byte.copy <file_table_nexp>,<output_file_sexp>, 115
 Byte.eof <file_table_nexp>, <lvar>, 114
 Byte.open {r|w|a}, <file_table_nvar>, <path_sexp>, 112
 Byte.position.get <file_table_nexp>, <position_nexp>, 114
 Byte.position.mark {{<file_table_nexp>}{, <marklimit_nexp>}}, 115
 Byte.position.set <file_table_nexp>, <position_nexp>, 114
 Byte.read.buffer <file_table_nexp>, <count_nexp>, <buffer_svar>, 114
 Byte.read.byte <file_table_nexp> {<nvar>}..., 112
 Byte.read.number <file_table_nexp> {<nvar>}..., 113
 Byte.truncate <file_table_nexp>,<length_nexp>, 115
 Byte.write.buffer <file_table_nexp>, <sexp>, 114
 Byte.write.byte <file_table_nexp> {{<nexp>}...{<sexp>}}, 113
 Byte.write.number <file_table_nexp> {<nexp>}..., 113

Call <user_defined_function>, 78
 CBRT(<nexp>), 59
 CEIL(<nexp>), 57
 CHR\$(<nexp>, ...), 64
 Clipboard.get <svar>, 134
 Clipboard.put <sexp>, 134
 CLOCK(), 63
 Cls, 97
 Console.front, 97
 Console.line.count <count_nvar>, 97
 Console.line.text <line_nexp>, <text_svar>, 97
 Console.line.touched <line_nvar> {, <press_lvar>}, 98
 Console.save <filename_sexp>, 98
 Console.title { <title_sexp>}, 98
 ConsoleTouch.resume, 90
 COS(<nexp>), 59
 COSH(<nexp>), 60
 D_U.break, 82
 D_U.continue, 81
 Debug.dump.array Array[], 93
 Debug.dump.bundle <bundlePtr_nexp>, 93
 Debug.dump.list <listPtr_nexp>, 93
 Debug.dump.scalars, 93
 Debug.dump.stack <stackPtr_nexp>, 93
 Debug.echo.off, 93
 Debug.echo.on, 92
 Debug.off, 92
 Debug.on, 92
 Debug.print, 93
 Debug.show, 95
 Debug.show.array Array[], 94
 Debug.show.bundle <bundlePtr_nexp>, 94
 Debug.show.list <listPtr_nexp>, 94
 Debug.show.program, 94
 Debug.show.scalars, 93
 Debug.show.stack <stackPtr_nexp>, 94
 Debug.show.watch, 94
 Debug.watch var, ..., 94
 DECODE\$(<charset_sexp>, <buffer_sexp>), 68
 DECODE\$(<type_sexp>, {<qualifier_sexp>}, <source_sexp>), 67
 Decrypt <pw_sexp>, <encrypted_svar>, <decrypted_svar>, 135
 Device <nexp>|<nvar>, 141
 Device <svar>, 141
 Device.Language <svar>, 141
 Device.Locale <svar>, 142
 Dialog.message {<title_sexp>}, {<message_sexp>}, <sel_nvar> {, <button1_sexp>{, <button2_sexp>{, <button3_sexp>}}}, 98
 Dialog.select <sel_nvar>, <Array\$[]>|<list_nexp>, {<title_sexp>}, 99

Dim Array [n, n, ...], Array\$(n, n, ...) ..., 40
 Do / Until <lexp>, 81
 Echo.off, 93
 Echo.on, 93
 Email.send <recipient_sexp>, <subject_sexp>, <body_sexp>, 130
 ENCODE\$(<charset_sexp>, <source_sexp>), 68
 ENCODE\$(<type_sexp>, {<qualifier_sexp>}, <source_sexp>), 66
 Encrypt {<pw_sexp>}, <source_sexp>, <encrypted_svar>, 134
 End{ <msg_sexp>}, 91
 ENDS_WITH(<sub_sexp>, <base_sexp>), 62
 Exit, 91
 EXP(<nexp>), 59
 F_N.break, 80
 F_N.continue, 80
 File.delete <lvar>, <path_sexp>, 106
 File.dir <path_sexp>, Array\$[] {, <dirmark_sexp>}, 107
 File.exists <lvar>, <path_sexp>, 107
 File.mkdir <path_sexp>, 107
 File.rename <old_path_sexp>, <new_path_sexp>, 107
 File.root <svvar>, 108
 File.size <size_nvar>, <path_sexp>, 108
 File.type <type_svar>, <path_sexp>, 108
 FLOOR(<nexp>), 57
 Fn.def name|name\$({nvar}|{svvar}|Array[]|Array\$[], ... {nvar}|{svvar}|Array[]|Array\$[]), 76
 Fn.end, 78
 Fn.rtn <sexp>|<nexp>, 77
 Font.clear, 95
 Font.delete {<font_ptr_nexp>}, 95
 Font.load <font_ptr_nvar>, <filename_sexp>, 95
 For <nvar> = <nexp> To <nexp> {Step <nexp>} / Next, 79
 FORMAT\$(<pattern_sexp>, <nexp>), 74
 FORMAT_USING\$(<locale_sexp>, <format_sexp> {, <exp>}...), 74
 FRAC(<nexp>), 57
 Ftp.cd <new_directory_sexp>, 126
 Ftp.close, 125
 Ftp.delete <filename_sexp>, 126
 Ftp.dir <list_nvar>, 126
 Ftp.get <source_sexp>, <destination_sexp>, 126
 Ftp.mkdir <directory_sexp>, 127
 Ftp.open <url_sexp>, <port_nexp>, <user_sexp>, <pw_sexp>, 125
 Ftp.put <source_sexp>, <destination_sexp>, 125
 Ftp.rename <old_filename_sexp>, <new_filename_sexp>, 126
 Ftp.rmdir <directory_sexp>, 126
 GETERROR\$(), 63
 GoSub <index_nexp>, <label>... / Return, 83
 GoSub <label> / Return, 83
 GoTo <index_nexp>, <label>..., 83
 GoTo <label>, 83

Gps.accuracy <nvar>, 195
 Gps.altitude <nvar>, 195
 Gps.bearing <nvar>, 195
 Gps.close, 191
 Gps.latitude <nvar>, 195
 Gps.location {{<time_nvar>}, {<prov_svar>}, {<count_nvar>}, {<acc_nvar>}, {<lat_nvar>}, {<long_nvar>}, {<alt_nvar>}, {<bear_nvar>}, {<speed_nvar>}}, 194
 Gps.longitude <nvar>, 195
 Gps.open {{<status_nvar>},{<time_nexp>},{<distance_nexp>}}, 191
 Gps.provider <svar>, 194
 Gps.satellites {{<count_nvar>}, {<sat_list_nexp>}}, 195
 Gps.speed <nvar>, 195
 Gps.status {{<status_var>}, {<infix_nvar>},{inview_nvar}, {<sat_list_nexp>}}, 191
 Gps.time <nvar>, 194
 Gr.arc <obj_nvar>, left, top, right, bottom, start_angle, sweep_angle, fill_mode, 163
 Gr.bitmap.create <bitmap_ptr_nvar>, width, height, 174
 Gr.bitmap.crop <new_bitmap_ptr_nvar>, <source_bitmap_ptr_nexp>, <x_nexp>, <y_nexp>, <width_nexp>, <height_nexp>, 175
 Gr.bitmap.delete <bitmap_ptr_nexp>, 175
 Gr.bitmap.draw <object_ptr_nvar>, <bitmap_ptr_nexp>, x, y, 176
 Gr.bitmap.drawinto.end, 177
 Gr.bitmap.drawinto.start <bitmap_ptr_nexp>, 176
 Gr.bitmap.fill <bitmap_ptr_nexp>, <x_nexp>, <y_nexp>, 176
 Gr.bitmap.load <bitmap_ptr_nvar>, <file_name_sexp>, 174
 Gr.bitmap.save <bitmap_ptr_nvar>, <filename_sexp>{, <quality_nexp>}, 176
 Gr.bitmap.scale <new_bitmap_ptr_nvar>, <bitmap_ptr_nexp>, width, height {, <smoothing_lexp> }, 175
 Gr.bitmap.size <bitmap_ptr_nexp>, width, height, 175
 Gr.bounded.touch touched, left, top, right, bottom, 168
 Gr.bounded.touch2 touched, left, top, right, bottom, 168
 Gr.brightness <nexp>, 162
 Gr.camera.autoshoot <bm_ptr_nvar>{, <flash_mode_nexp> {, focus_mode_nexp} }, 180
 Gr.camera.manualShoot <bm_ptr_nvar>{, <flash_mode_nexp> {, focus_mode_nexp} }, 180
 Gr.camera.select 1|2, 179
 Gr.camera.shoot <bm_ptr_nvar>, 179
 Gr.circle <obj_nvar>, x, y, radius, 164
 Gr.clip <object_ptr_nexp>, <left_nexp>,<top_nexp>, <right_nexp>, <bottom_nexp>{, <RO_nexp>}, 184
 Gr.close, 162
 Gr.cls, 161
 Gr.color {{alpha}{, red}{, green}{, blue}{, style}{, paint}}, 158
 Gr.front flag, 162
 Gr.get.bmpixel <bitmap_ptr_nvar>, x, y, alpha, red, green, blue, 176
 Gr.get.params <object_ptr_nexp>, <param_array\$[]>, 181
 Gr.get.pixel x, y, alpha, red, green, blue, 181
 Gr.get.position <object_ptr_nexp>, x, y, 181
 Gr.get.textbounds <exp>, left, top, right, bottom, 172
 Gr.get.type <object_ptr_nexp>, <type_svar>, 181
 Gr.get.value <object_ptr_nexp> {, <tag_sexp>, <value_nvar | value_svar>}..., 182
 Gr.getDL <dl_array[]> {, <keep_all_objects_lexp> }, 185

Gr.group <object_number_nvar>{, <obj_nexp>}..., 166
 Gr.group.getDL <object_number_nvar>, 166
 Gr.group.list <object_number_nvar>, <list_ptr_nexp>, 166
 Gr.group.newDL <object_number_nvar>, 166
 Gr.hide <object_number_nexp>, 167
 Gr.line <obj_nvar>, x1, y1, x2, y2, 163
 Gr.modify <object_ptr_nexp> {, <tag_sexp>, <value_nexp | value_sexp>}..., 182
 Gr.move <object_ptr_nexp>{{, dx}{, dy}}, 181
 Gr.newDL <dl_array[{<start>,<length>}]>, 185
 Gr.onGrTouch.resume, 168
 Gr.open {{alpha}{, red}{, green}{, blue}{, <ShowStatusBar_lexp>}{, <Orientation_nexp>}}, 158
 Gr.orientation <nexp>, 160
 Gr.oval <obj_nvar>, left, top, right, bottom, 163
 Gr.paint.copy {{<src_nexp>}{, <dst_nexp>}}, 177
 Gr.paint.get <object_ptr_nvar>, 177
 Gr.paint.reset {<nexp>}, 178
 Gr.point <obj_nvar>, x, y, 162
 Gr.poly <obj_nvar>, list_pointer {x,y}, 164
 Gr.rect <obj_nvar>, left, top, right, bottom, 163
 Gr.render, 160
 Gr.rotate.end {<obj_nvar>}, 178
 Gr.rotate.start angle, x, y{<obj_nvar>}, 178
 Gr.save <filename_sexp> {,<quality_nexp>}, 181
 Gr.scale x_factor, y_factor, 161
 Gr.screen width, height{, density }, 161
 Gr.screen.to_bitmap <bm_ptr_nvar>, 180
 Gr.set.antialias {{<lexp>}{,<paint_nexp>}}, 159
 Gr.set.pixels <obj_nvar>, pixels[{<start>,<length>}] {x,y}, 164
 Gr.set.stroke {{<nexp>}{,<paint_nexp>}}, 159
 Gr.show <object_number_nexp>, 167
 Gr.show.toggle <object_number_nexp>, 167
 Gr.statusbar <height_nvar> {, showing_lvar}, 160
 Gr.statusbar.show <nexp>, 160
 Gr.text.align {{<type_nexp>}{,<paint_nexp>}}, 169
 Gr.text.bold {{<lexp>}{,<paint_nexp>}}, 169
 Gr.text.draw <object_number_nvar>, <x_nexp>, <y_nexp>, <text_object_sexp>, 173
 Gr.text.height <height_nvar> {, <up_nvar> } {, <down_nvar>}, 171
 Gr.text.setfont {{<font_ptr_nexp>|<font_family_sexp> } {, <style_sexp> } {,<paint_nexp>}}, 170
 Gr.text.size {{<size_nexp>}{,<paint_nexp>}}, 169
 Gr.text.skew {{<skew_nexp>}{,<paint_nexp>}}, 169
 Gr.text.strike {{<lexp>}{,<paint_nexp>}}, 170
 Gr.text.typeface {{<nexp> } {, <style_nexp> } {,<paint_nexp>}}, 171
 Gr.text.underline {{<lexp>}{,<paint_nexp>}}, 170
 Gr.text.width <nvar>, <exp>, 172
 Gr.touch touched, x, y, 167
 Gr.touch2 touched, x, y, 168
 GR_COLLISION(<object_1_nvar>, <object_2_nvar>), 62
 GrabFile <result_svar>, <path_sexp>{, <unicode_flag_lexp>}, 111

GrabURL <result_svar>, <url_sexp>{, <timeout_nexp>}, 111
 Headset <state_nvar>, <type_svar>, <mic_nvar>, 147
 HEX\$(<nexp>), 69
 HEX(<sexp>), 61
 Home, 145
 Html.clear.cache, 120
 Html.clear.history, 120
 Html.close, 120
 Html.get.datalink <data_svar>, 119
 Html.go.back, 120
 Html.go.forward, 120
 Html.load.string <html_sexp>, 118
 Html.load.url <file_sexp>, 118
 Html.open {<ShowStatusBar_lexp> {, <Orientation_nexp>}}, 117
 Html.orientation <nexp>, 118
 Html.post <url_sexp>, <list_nexp>, 119
 Http.post <url_sexp>, <list_nexp>, <result_svar>, 120
 HYPOT(<nexp_x>, <nexp_y>), 59
 If / Then / Else, 79
 If / Then / Else / Elseif / Endif, 78
 Include FilePath, 84
 Inkey\$ <svar>, 100
 Input {<prompt_sexp>, <result_var>{, {<default_exp>}{<canceled_nvar>}}, 100
 INT\$(<nexp>), 69
 INT(<nexp>), 57
 IS_IN(<sub_sexp>, <base_sexp>{, <start_nexp>}, 61
 IS_NUMBER(<sexp>), 60
 Join <source_array\$[]>, <result_svar> {, <separator_sexp>{, <wrapper_sexp>}}, 136
 Join.all <source_array\$[]>, <result_svar> {, <separator_sexp>{, <wrapper_sexp>}}, 136
 Kb.hide, 102
 Kb.resume, 103
 Kb.show, 102
 Kb.showing <lvar>, 103
 Kb.toggle, 103
 Key.resume, 91
 LEFT\$(<sexp>, <count_nexp>), 64
 LEN(<sexp>), 61
 Let, 55
 List.add <pointer_nexp>{, <exp>}..., 45
 List.add.array numeric_list_pointer, Array[{<start>,<length>}], 46
 List.add.array string_list_pointer, Array\$[{<start>,<length>}], 46
 List.add.list <destination_list_pointer_nexp>, <source_list_pointer_nexp>, 46
 List.clear <pointer_nexp>, 47
 List.create N|S, <pointer_nvar>, 45
 List.get <pointer_nexp>, <index_nexp>, <var>, 47
 List.insert <pointer_nexp>, <index_nexp>, <sexp>|<nexp>, 46
 List.remove <pointer_nexp>,<index_nexp>, 46
 List.replace <pointer_nexp>, <index_nexp>, <sexp>|<nexp>, 46

List.search <pointer_nexp>, value|value\$, <result_nvar>{,<start_nexp>}, 47
 List.size <pointer_nexp>, <nvar>, 47
 List.toArray <pointer_nexp>, Array\$[] | Array[], 47
 List.type <pointer_nexp>, <svar>, 47
 LOG(<nexp>), 59
 LOG10(<nexp>), 59
 LOWER\$(<sexp>), 69
 LowMemory.resume, 91
 MAX(<nexp>, <nexp>), 57
 MenuKey.resume, 91
 MID\$(<sexp>, <start_nexp>{, <count_nexp>}), 64
 MIN(<nexp>, <nexp>), 57
 mkdir <path_sexp>, 107
 MOD(<nexp1>, <nexp2>), 58
 MyPhoneNumber <svar>, 130
 Notify <title_sexp>, <subtitle_sexp>, <alert_sexp>, <wait_lexp>, 147
 OCT\$(<nexp>), 69
 OCT(<sexp>), 61
 OnBackground:, 145
 OnBackKey:, 90
 OnBtReadReady:, 129
 OnConsoleTouch:, 90
 OnError:, 89
 OnGrTouch:, 168
 OnKbChange:, 103
 OnKeyPress:, 91
 OnLowMemory:, 91
 OnMenuKey:, 90
 OnTimer:, 133
 Pause <ticks_nexp>, 148
 Phone.call <sexp>, 130
 Phone.dial <sexp>, 131
 Phone.info <nexp>|<nvar>, 142
 Phone.rcv.init, 131
 Phone.rcv.next <state_nvar>, <number_svar>, 131
 PI(), 59
 Popup <message_sexp>{,<x_nexp>}{,<y_nexp>}{,<duration_lexp>}}, 101
 POW(<nexp1>, <nexp2>), 59
 Print {<exp> {,|;}} ..., 96
 Program.info <nexp>|<nvar>, 85
 RANDOMIZE({<nexp>}), 57
 Read.data <number>|<string> {,<number>|<string>...,<number>|<string>}, 91
 Read.from <nexp>, 92
 Read.next <var>, ..., 92
 Rem, 52
 REPLACE\$(<sexp>, <argument_sexp>, <replace_sexp>), 65
 RIGHT\$(<sexp>, <count_nexp>), 65
 Ringer.get.mode <nvar>, 135

Ringer.get.volume <vol_nvar> { , <max_nvar>, 135
 Ringer.set.mode <nexp>, 135
 Ringer.set.volume <nexp>, 135
 RND(), 57
 ROUND(<value_nexp>{, <count_nexp>{, <mode_sexp>{}}), 58
 Run <filename_sexp>{, <data_sexp>}, 85
 Screen.rotation, size[], realsize[], density, 143
 Screen.rotation <nvar>, 144
 Screen.size size[], realsize[], density, 144
 Select <sel_nvar>, <Array\$[]>|<list_nexp>, {,<title_sexp> {, <message_sexp> } } {,<press_lvar> }, 101
 Sensors.close, 197
 Sensors.list <sensor_array\$[]>, 196
 Sensors.open <type_nexp>{:<delay_nexp>}{, <type_nexp>{:<delay_nexp>}, ...}, 196
 Sensors.read sensor_type, p1, p2, p3, 197
 SGN(<nexp>), 56
 SHIFT(<value_nexp>, <bits_nexp>), 61
 SIN(<nexp>), 59
 SINH(<nexp>), 59
 Sms.rcv.init, 131
 Sms.rcv.next <svar>, 131
 Sms.send <number_sexp>, <message_sexp>, 131
 Socket.client.close, 123
 Socket.client.connect <server_sexp>, <port_nexp> { , <wait_lexp> }, 121
 Socket.client.read.file <file_nexp>, 122
 Socket.client.read.line <line_svar>, 122
 Socket.client.read.ready <nvar>, 122
 Socket.client.server.ip <svar>, 122
 Socket.client.status <status_nvar>, 122
 Socket.client.write.bytes <sexp>, 123
 Socket.client.write.file <file_nexp>, 123
 Socket.client.write.line <line_sexp>, 122
 Socket.myIP <array\$[]>{, <nvar>}, 123
 Socket.myIP <svar>, 123
 Socket.server.client.ip <nvar>, 125
 Socket.server.close, 125
 Socket.server.connect {<wait_lexp>}, 123
 Socket.server.create <port_nexp>, 123
 Socket.server.disconnect, 125
 Socket.server.read.file <file_nexp>, 125
 Socket.server.read.line <svar>, 124
 Socket.server.read.ready <nvar>, 124
 Socket.server.status <status_nvar>, 124
 Socket.server.write.bytes <sexp>, 124
 Socket.server.write.file <file_nexp>, 125
 Socket.server.write.line <line_sexp>, 124
 Soundpool.load <soundID_nvar>, <file_path_sexp>, 189
 Soundpool.open <MaxStreams_nexp>, 189
 Soundpool.pause <streamID_nexp>, 190

Soundpool.play <streamID_nvar>, <soundID_nexp>, <rightVolume_nexp>, <leftVolume_nexp>,
 <priority_nexp>, <loop_nexp>, <rate_nexp>, 189
 Soundpool.release, 190
 Soundpool.resume <streamID_nexp>, 190
 Soundpool.setpriority <streamID_nexp>, <priority_nexp>, 190
 Soundpool.setrate <streamID_nexp>, <rate_nexp>, 190
 Soundpool.setvolume <streamID_nexp>, <leftVolume_nexp>, <rightVolume_nexp>, 190
 Soundpool.stop <streamID_nexp>, 190
 Soundpool.unload <soundID_nexp>, 189
 Split <result_array\$[]>, <sexp> {, <test_sexp>}, 136
 Split.all <result_array\$[]>, <sexp> {, <test_sexp>}, 136
 Sql.close <DB_pointer_nvar>, 150
 Sql.delete <DB_pointer_nvar>, <table_name_sexp>{<where_sexp>{<count_nvar>}}, 152
 Sql.drop_table <DB_pointer_nvar>, <table_name_sexp>, 150
 Sql.exec <DB_pointer_nvar>, <command_sexp>, 153
 Sql.insert <DB_pointer_nvar>, <table_name_sexp>, C1\$, V1\$, C2\$, V2\$, ...,CN\$, VN\$, 150
 Sql.new_table <DB_pointer_nvar>, <table_name_sexp>, C1\$, C2\$, ...,CN\$, 150
 Sql.next doneFlag, cursorVar {{, svar}...{, array\$[] {, nColVar}}}, 152
 Sql.open <DB_pointer_nvar>, <DB_name_sexp>, 150
 Sql.query <cursor_nvar>, <DB_pointer_nvar>, <table_name_sexp>, <columns_sexp> {, <where_sexp> {,
 <order_sexp>} }, 151
 Sql.query.length <length_nvar>, <cursor_nvar>, 151
 Sql.query.position <position_nvar>, <cursor_nvar>, 151
 Sql.raw_query <cursor_nvar>, <DB_pointer_nvar>, <query_sexp>, 153
 Sql.update <DB_ptr_nvar>, <table_name_sexp>, C1\$, V1\$, C2\$, V2\$,...,CN\$, VN\${: <where_sexp>}, 153
 SQR(<nexp>), 59
 Stack.clear <ptr_nexp>, 51
 Stack.create N|S, <ptr_nvar>, 51
 Stack.isEmpty <ptr_nexp>, <nvar>, 51
 Stack.peek <ptr_nexp>, <nvar>|<svar>, 51
 Stack.pop <ptr_nexp>, <nvar>|<svar>, 51
 Stack.push <ptr_nexp>, <nexp>|<sexp>, 51
 Stack.type <ptr_nexp>, <svar>, 51
 STARTS_WITH(<sub_sexp>, <base_sexp>{, <start_nexp>}, 62
 STR\$(<nexp>), 69
 STT.listen, 138
 STT.results <string_list_ptr_nexp>, 139
 Su.close, 199
 Su.open, 198
 Su.read.line <svar>, 198
 Su.read.ready <nvar>, 198
 Su.write <sexp>, 198
 Sw.begin <exp>, 87
 Sw.break, 88
 Sw.case <exp>, ..., 88
 Sw.case <op><exp>, 88
 Sw.default, 88
 Sw.end, 88

Swap <nvar_a>|<svar_a>, <nvar_b>|<svar_b>, 148
 System.close, 198
 System.open, 198
 System.read.line <svar>, 198
 System.read.ready <nvar>, 198
 System.write <ssexp>, 198
 TAN(<nexp>), 59
 Text.close <file_table_nexp>, 109
 Text.eof <file_table_nexp>, <lvar>, 110
 Text.input <svar>{, { <text_ssexp> }, <title_ssexp> }, 101
 Text.open {r|w|a}, <file_table_nvar>, <path_ssexp>, 108
 Text.position.get <file_table_nexp>, <position_nvar>, 110
 Text.position.mark {{<file_table_nexp>}{, <marklimit_nexp>}}, 111
 Text.position.set <file_table_nexp>, <position_nexp>, 110
 Text.readln <file_table_nexp> {,<svar>}..., 109
 Text.writeln <file_table_nexp>, {<exp> {,|;}} ..., 110
 TGet <result_svar>, <prompt_ssexp> {, <title_ssexp>}, 102
 Time {<time_nexp>,<Year\$,<Month\$,<Day\$,<Hour\$,<Minute\$,<Second\$,<WeekDay,<isDST, 132
 TIME(), 63
 TIME(<year_exp>, <month_exp>, <day_exp>, <hour_exp>, <minute_exp>, <second_exp>), 63
 Timer.clear, 134
 Timer.resume, 133
 Timer.set <interval_nexp>, 133
 TimeZone.get <tz_svar>, 133
 TimeZone.list <tz_list_pointer_nexp>, 133
 TimeZone.set { <tz_ssexp> }, 133
 TODEGREES(<nexp>), 60
 Tone <frequency_nexp>, <duration_nexp>{, <duration_chk_lexp>, 148
 TORADIANS(<nexp>), 60
 TRIM\$(<ssexp>{, <test_ssexp>}), 65
 TTS.init, 138
 TTS.speak <ssexp> {, <wait_lexp>}, 138
 TTS.speak.toFile <ssexp> {, <path_ssexp>}, 138
 TTS.stop, 138
 UCODE(<ssexp>{, <index_nexp>}), 61
 UnDim Array[], Array\$[], ..., 41
 UPPER\$(<ssexp>), 69
 USING\$({<locale_ssexp>}, <format_ssexp> {, <exp>}...), 70
 VAL(<ssexp>), 60
 VERSION\$(), 69
 Vibrate <pattern_array[{<start>,<length>}]>,<nexp>, 148
 VolKeys.off, 149
 VolKeys.on, 149
 W_R.break, 81
 W_R.continue, 81
 WakeLock <code_nexp>{, <flags_nexp>, 145
 While <lexp> / Repeat, 80
 WiFi.info {{<SSID_svar>}{, <BSSID_svar>}{, <MAC_svar>}{, <IP_var>}{, <speed_nvar>}}, 144

WifiLock <code_nexp>, 146
WORD\$(<source_sexp>, <n_nexp> {, <test_sexp>}), 66
Zip.close, <file_table_nexp>, 116
Zip.count <path_sexp>, <nvar>, 116
Zip.dir <path_sexp>, Array\$[] {,<dirmark_sexp>}, 116
Zip.open {r|w|a}, <file_table_nvar>, <path_sexp>, 116
Zip.read <file_table_nexp>,<buffer_svar>, <file_name_sexp>, 116, 117

Appendix B – Sample Programs

The programs are loaded into "<pref base drive>/rfo-basic/source/Sample_Programs" when a new release of BASIC! is installed. You can access them by selecting Menu->Load. Tap the "Sample_Programs" lines. The sample programs will be listed and can be loaded.

If you load and save one of these programs, the program will be saved in "<pref base drive>/rfo-basic/source/" not in "<pref base drive>/rfo-basic/source/Sample_Programs"

You can force BASIC! to re-load these programs by:

- Select Menu->Delete
- Navigate to "rfo-basic/source/Sample_Programs/"
- Delete the "f01_vxx.xx_read_me file"
- Exit BASIC! using Menu->Exit or Menu->More->Exit
- Re-enter BASIC!

Appendix C – Launcher Shortcut Tutorial

Introduction

This tutorial will "compile" a BASIC! program and create an "application" that resides on your Android device home page. This "application" will have its own Icon and Name. The official Android name for this type of "application" is "Shortcut." The BASIC! application must be installed for this to work.

There is also an option to actually build a standalone application .apk file that does not require the BASIC! application to be installed. The process is more difficult but it will result in an application that can be offered on the Google Play Store. See Appendix D.

How to Make a Shortcut Application (older versions of Android—prior to Android 4.0)

1. Start BASIC!
2. Select Menu->Exit or Menu->More->Exit to Exit BASIC!
3. Do a long press on the HOME screen.
4. You should see something like the following:



5. Tap Shortcuts.
6. Scroll down the Select Shortcut page until you see the BASIC! icon with the Launcher Shortcuts Label.



7. Tap the BASIC! Icon.

8. This screen will appear:



9. Fill out the Form exactly as shown.

- The Program File Name is Sample_Programs/f13_animations.bas
- The Icon File Name is cartman.png
- The Shortcut Name is Cartman

10. Tap OK.



11. You should see something like this on your HOME screen:

12. Tap the Cartman Shortcut.

13. BASIC! will start and run the Cartman Jumping Demo.

How to Make a Shortcut Application (newer versions of Android—Android 4.0 and later)

1. Select the Apps Page.

2. At the top of the screen, tap Widgets.

3. Scroll horizontally until you see the BASIC! icon that says Launcher Shortcuts.
4. Touch and hold that entry. It will be moved to the Home page.
5. Continue with step 8, above.

What you need to know

- The icon image file must be located in the "<pref base drive>/rfo-basic/data/" directory.
- The program that you are going to run must be in the "source" directory or one of its sub directories. In this example, the file was located in the Sample_Programs(d) subdirectory of the "source(d)" directory.
- The icon should be a .png file. A Google search for "icon" will reveal thousands for free icons. Just copy your icon into "rfo-basic/data" on the SD card.
- Be very careful to correctly spell the names of the program and icon files. BASIC! does not check to see if these files actually exist during the "compile" process. If you enter the name of an icon file that does not exist, your shortcut will have the generic Android icon. If the file name you specified does not exist, when you tap the Shortcut you will see an error message in the form of program file in the Editor.
- The Shortcut name should be nine (9) characters or less. Android will not show more than nine characters.
- You can create as many shortcuts as you home screen(s) can handle.
- Tapping "Cancel" in the Launcher Shortcuts dialog will simply cancel the operation and return to the home screen.
- If you plan to use a BASIC! Launcher Shortcut, you should always exit BASIC! using Exit-Menu or Menu->More->Exit. If a Launched program is running, tapping BACK once or twice will exit BASIC back to the Home Screen.

Appendix D – Building a Standalone Application

Note: BASIC! collaborator Nicolas Mougin, has created a suite of automated tools for generating standalone applications. These tools can be downloaded from this website:

<http://rfo-basic.com>

Using Mr. Mougin's tools, you avoid having to do all of the following.

If you have any questions or problems with this tool you can contact Mr. Mougin and other users of the tool at the BASIC! forum in this thread:

<http://rfobasic.freeforums.org/rfo-basic-app-builder-f20.html>

Introduction

This document will demonstrate how to create a standalone application from a BASIC! program. The resulting application does not need to have BASIC! installed to run. It will have its own application name and application icon. It may be distributed in the Google Play Store or elsewhere. The process involves setting up the Android development environment and making some simple, directed changes to the BASIC! Java source code. Since Google changes the Development Environment every so often, this procedure does not necessarily reflect the latest version of the ADT.

License Information

BASIC! is distributed under the terms of the [GNU GENERAL PUBLIC LICENSE](#). The license requires that the source code for "derivative works" be made available to anyone who asks. The author of BASIC! interprets this to mean that the license applies only to derivatives of the BASIC! *source interpreter code*. It does not apply to source code for BASIC! applications, i.e., code that you have written using the BASIC! language.

Before You Start

Run the sample program, **source/my_program.bas**.

We are going to turn this program into a practice APK.

Setting Up the Development Environment

These instructions are based on using the Android Developer Tools (ADT) plug-in for the Eclipse Integrated Development Environment (IDE) to build an Android application.

This is no longer the recommended toolset. Android Studio is now the official IDE for Android. For information about Android Studio, visit Android's Developer Tools page at

<http://developer.android.com/sdk/index.html>.

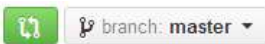

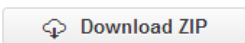
These instructions will be updated for the new tools in a future version of this manual.

1. Download and install the latest version of the Java Development Kit (JDK). Find this by Googling "java jdk download" and going to the listed oracle.com download site. Do not download from any other site. Note: The JDK download includes the Java Run Time Environment (JRE) which is also needed.
2. Download and install the "Eclipse IDE for Java Developers" from <https://eclipse.org/downloads/>. The current version is 4.4.1 "Luna". You may prefer to download and install an older version from <https://eclipse.org/kepler/> or <https://eclipse.org/juno/>, because these instructions were written from "Juno".
3. Visit Android's "Installing the Eclipse Plugin" page at <http://developer.android.com/sdk/installing/installing-adt.html>. From there you can download the Android Developer Tools (ADT) Plugin. Follow the installation instructions found there.

The instructions below have been changed over time for different versions, most recently for Eclipse 4.2 (Juno) and ADT 22.2, and you may find differences in the version you have installed. Please visit the BASIC! users' forum at <http://rfobasic.freeforums.org> to report any problems you find.

4. BASIC! supports most versions of Android, which can lead to spurious build errors. You may need to change API level checking from Error to Warning:
 - a. Select **Window->Preferences->Android->Lint Error Checking**
 - b. Find **NewApi** and click on it.
 - c. In the dropdown list, change the Severity from Error to Warning.
 - d. Click **Apply**

Download the BASIC! Source Code from the GitHub Repository

1. Go to: <https://github.com/RFO-BASIC/Basic>.
2. Find the branch/tag selection button:  **Basic** /  Click the button and the "Tags" tab to select the tagged release version you want. If you do not select a tag, you will get the source for the latest development build. The latest development source is not guaranteed to be stable.
3. Click the "Download ZIP" button:  found on the right side of the page. The name of the downloaded ZIP file depends on which tag you selected in the previous step.
4. Put the unzipped source files into a folder. For this exercise, name the folder "Cats". You should use a different folder for each new APK that you create.

– OR –

Download the BASIC! Source Code from the Legacy Archive

1. Go to: <http://laughton.com/basic/versions/index.html> and select the version number you want. For versions later than v01.77, you must download the source from GitHub.
2. Look for the section heading, "**Download BASIC! Source Code.**" Click on the "**here**" to download "Basic.zip".

- Older versions of this file contain only the BASIC! source code. More recent versions also contain a copy of the Android-loadable Basic.apk file and a copy of this manual.
- Put the unzipped source files into a folder. For this exercise, name the folder "Cats". You should use a different folder for each new APK that you create.

Note: In the remainder of this tutorial, we assume our application is about cats, thus we are using the name "Cats". For your own APK, you should choose a name that matches your application.

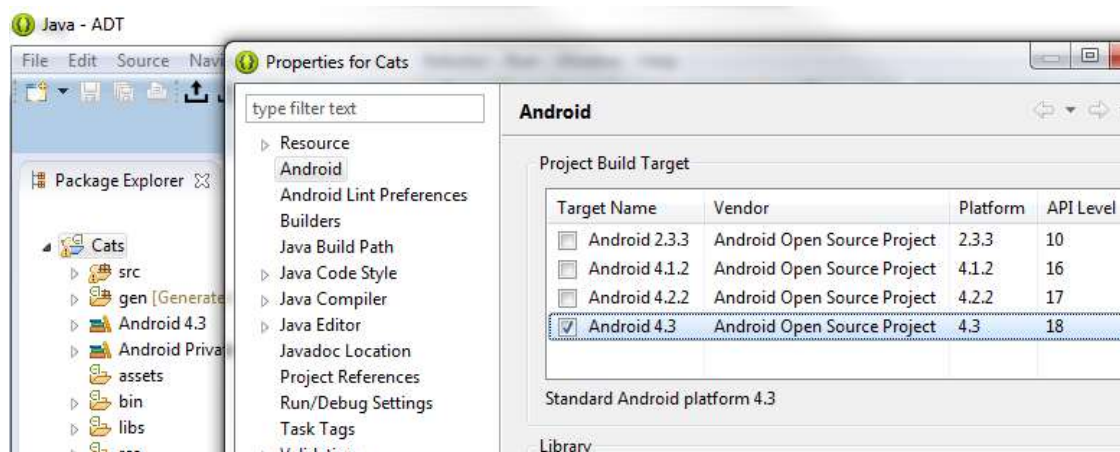
Create a New Project in Eclipse

Note: to simplify later steps, turn off the automatic build feature of Eclipse before creating the new project:

- On the menu, click **Project**.
- Uncheck **Build Automatically**.

Now you can create the project.

- Select: **File -> New -> Project...->Android->Android Project from Existing Code**.
- In the **Import Projects** dialog box, browse to the folder where you put the BASIC! source files.
- Under **Projects to Import**, click the project you are importing. Make sure it is checked.
- Under **New Project Name**, change the project name from "Basic" to "Cats". Click **Finish**.
- In the **Package Explorer** window on the left, right-click on **Cats** and select **Properties** (near the bottom of the list).
- In the Properties dialog box, select **Android**:



- Check the highest level of the Android OS available in this list. Do this without regard to the level of OS in your device(s). (Note for KitKat users: BASIC! is currently compiled with API 18. The build works with higher levels, but you may choose to limit the OS level to 4.3, API 18.)
- Click **Apply** then **OK**.

The Basic source is now ready for making an APK.

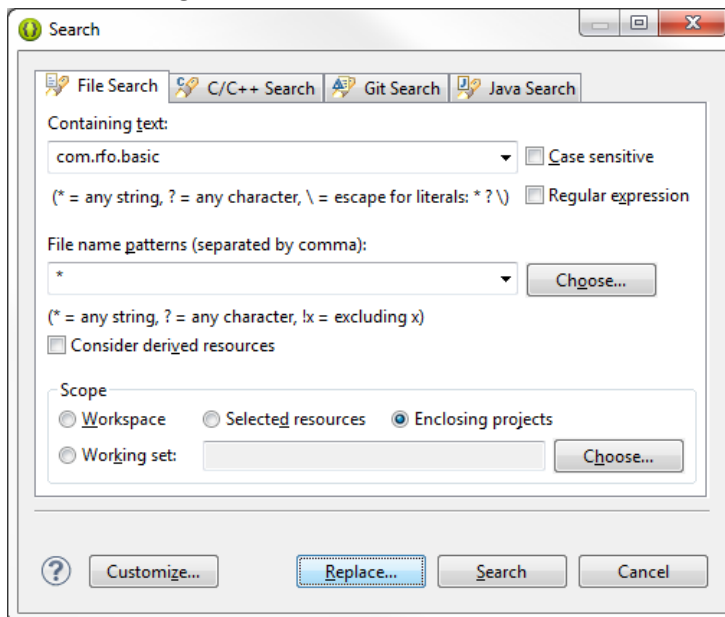
Rename the Package

The package name is what makes your application different from every other application that runs on Android devices. No matter what you name your application, it is the package name that Android uses to identify your particular application.

In Eclipse

First, use the search-and-replace dialog to update references to the package name.

1. From the Menu Bar, select: **Search->File**.
2. Fill in the dialog like this:

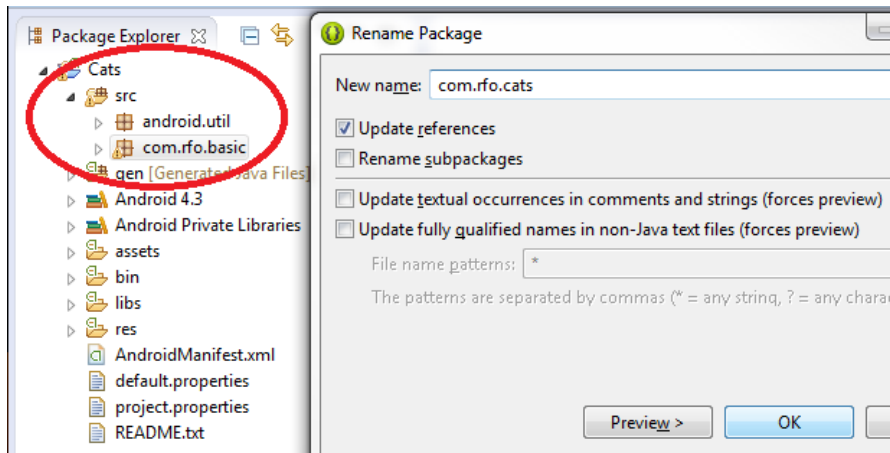


3. Click **Replace**. The Replace Text Matches Dialog Box opens.
4. Enter "com.rfo.cats" in the **With:** field, and click **OK**.

Next, tell Eclipse about the package change.

5. In the **Package Explorer**, click and open the **Cats -> src** hierarchy as shown below.
6. Select the **com.rfo.basic** package.
7. From the Menu Bar, select **File->Rename** to open the **Rename Package** dialog.
8. In the **New name:** field, enter "com.rfo.cats".

9. Make sure the **Update references** box is checked and click **OK**.



Your project is ready to build. Restore the automatic build feature of Eclipse:

10. Click **Project->Build Automatically**. Make sure the checkmark appears.

The project should build without errors.

Other IDEs

Most changes in this Appendix are simply changes in source files, and so they apply to any IDE. Editing the package name is a more complex. Eclipse hides some of the complexity, but it also requires special actions when renaming the package so the IDE knows about the change. A different IDE, it may have its own requirements.

This section will describe all of the changes necessary to rename the package. Your IDE may do them, or you may have to do them yourself with a text editor. Some editors can do global search-and-replace across an entire file tree, which can save you a lot of work.

1. The package name is also a path name. If all of your source files are in a directory called "**src**", and your package name is "**com.rfo.basic**", then your source files are really in a directory called `src/com/rfo/basic`. If you change the package, for example, to "**com.rfo.cats**", then you must also move all of the source files to the new directory, in this case `src/com/rfo/cats`.
2. Everywhere it occurs, the string "**com.rfo.basic**" must be changed to your new package name.
 - a. The Android manifest file (**AndroidManifest.xml**) contains the line
`package="com.rfo.basic"`
If you have a `bin` directory in your project, it may contain a copy of the manifest file. It was generated by the Eclipse build process. You may ignore it or delete it.
 - b. Two layout files, **res/layout/editor.xml** and **res/layout/console.xml**, contain `class` attributes:
`class="com.rfo.basic.Editor$LinedEditText"`
`class="com.rfo.basic.Run$ConsoleListView"`
 - c. Every `.java` file in the source directory (`*.java`) contains the line
`package com.rfo.basic;`

- d. Several .java files (currently 7) files contain a line that starts like one of these:

```
import com.rfo.basic.  
import static com.rfo.basic.
```

Some files have more than one such line.
- e. Two files, **Editor.java** and **LauncherShortcuts.java**, contain one of these strings:

```
public static final String EXTRA_LS_FILENAME = "com.rfo.basic.fn";  
public static final String EXTRA_LS_FILENAME = "com.rfo.basic.doRestart";
```

You may safely ignore these, since they are not used in a standalone APK. However, it does not hurt to change these instances of `com.rfo.basic` along with all the others.

Renaming complete

At this point the package has been successfully renamed. Next we will create a practice APK that you then use to make your own APK.

Modifications to setup.xml

You can customize your apk with settings in the resource file **setup.xml**. Some changes are required for any apk.

In the **Package Explorer**, expand **Cats/res/values** and then double-click **setup.xml**. The file opens in the window on the right. There are tabs at the bottom of the window that let you change how the file is displayed. If the **Resources** tab is selected, you see a special Eclipse Resource Editor. This editor does not know how to modify all of the properties in this file. Click on the **setup.xml** tab to display the actual XML text, as shown in the image below. In this view, you can also see the comments describing the values you can change for your application.

Be sure you change only values, not names. Names are shown as blue text in quotes. Values are shown as black text with no quotation marks. If you change a name, Java can not find the item to get its value.

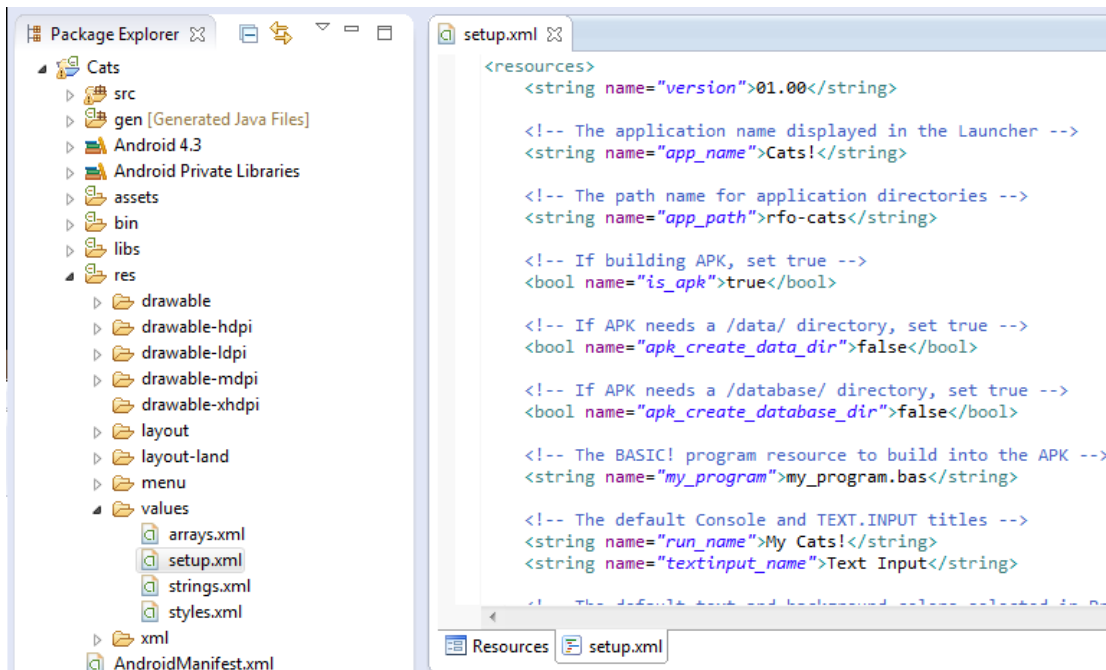
Change the value of "**app_name**" from **BASIC!** to **Cats!**. This is the name of the application as it will appear on your Android device.

Change the value of "**app_path**" to **rfo-cats**. This will be the directory on the SD card where your files are stored, if you choose to have a directory for files for your application. Make this change even if you do not choose to create directories for your application. It has implications in other parts of the code.

Change the value of "**is_apk**" to **true**.

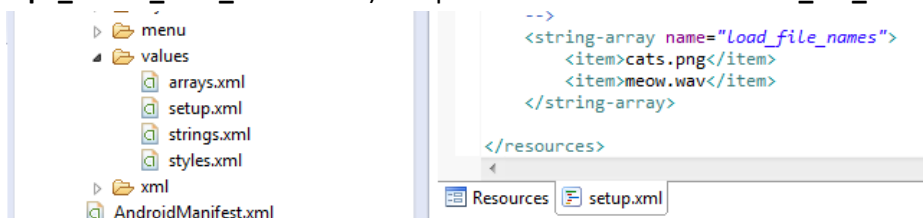
The items "**apk_create_data_dir**" and "**apk_create_database_dir**" are flags that control whether directories are created under "**app_path**" for your application's files. Since this practice application does not need any directories, change the values of both to **false**.

The item "**load_file_names**" is a list of files that you want loaded to the SD card. This practice application uses the sound clip **meow.wav**. Running under standard BASIC!, the program would use the file **rfo-basic/data/meow.wav**. As a standalone application, it can use a file image built into APK. Since the practice application is not using a real file, you can leave the "**load_file_names**" list empty.



Save the changes to setup.xml.

Note: If you do want to load files to the SD card, you must allow the creation of the data directory (set "apk_create_data_dir" to true) and put the file names in the "load_file_names" list like this:



The files will be loaded from the assets folder. This will be explained in the next sections.

At this point, you have modified the source files of the BASIC! interpreter so it can be packaged into a standalone application. You can build and run it, if you like, but it will display a blank Console. That's result you get when you build BASIC! into an application but you don't give it a program to run.

Advanced Customization with setup.xml

The **setup.xml** file allows more customization than the minimum required for **Cats**.

The VERSION\$() function gets its value from the item named "**version**". If your application uses VERSION\$(), this is where you set the value you want it to return.

The item "**my_program**" is the name of the BASIC! program you want your application to run. This will be explained in the next section.

If the **.bas** program files in the **source** subdirectory of your **assets** are encrypted, you must set the item **"apk_programs_encrypted"** to **true**. This tells BASIC! that it must decrypt programs as it loads them.

The item **"run_name"** is the default title of the Console. Your program can set the Console title, but you may want to change the default title here. Similarly, **"select_name"** and **"textinput_name"** set the names of the screens for the **Select**, **Text.Input**, and **TGet** commands.

If your app uses Console output, you can set the text colors here. The mapping of **"color1"**, **"color2"**, and **"color3"** to text foreground, background, and underline depend on your **Screen Colors** setting under **Preferences**.

As mentioned earlier, the **"load_file_names"** is a list of files that you want loaded to the SD card. Any files listed here will be copied from the **assets** directory to the SD card every time your app starts. This may be the internal Android file system or a file system mounted on an external SD card. After that, your app will access the files from the SD card file system, not from **assets**. Any changes your program makes to these files will be overwritten the next time the app starts. To preserve changes, your program must copy or rename the file. The copied/renamed file will not be overwritten when your program runs again.

Several items in **setup.xml** control the splash screen and file-load progress indicator. If you don't change anything, your **Cats** app will display the default BASIC! splash screen briefly. You may not see it because the apk does not load any files.

You can turn off the splash screen by setting **"splash_display"** to **false**.

The splash screen image is the resource file **res/drawable/splash.png**. To customize the splash screen, replace this file in your project with an image of your own. Your image file must be named **splash.jpg**.

If you do not use a splash screen, you may remove this file from your project to make your apk a little smaller. Do not remove the file **res/drawable/blank.jpg**.

You can also customize the file-load progress indicator. Progress is shown in a pop-up over the splash screen. The title comes from the string **"loading_msg"**. Each increment of progress is a copy of the string **"progress_marker "**. The default string is a single period (.).

You can turn off the progress indicator by making the **"loading_msg"** an empty string.

The **setup.xml** file is intended to collect the fields you are most likely to want to change all in one place. There are many other strings and values that can be set to customize your app. Another file with fields you may find interesting is **strings.xml**, where default values of many strings are set. For example, you can change the default title string for the Speech-to-Text dialog by editing the string named **sst_prompt**.

Later, we will describe how to change the default values of Preference options in the file **settings.xml**.

Files and Resources

Standard BASIC! loads its sample programs and the data files they need from the **assets** folder of the Eclipse project. Android treats the **assets** folder like a file system. At startup, BASIC! simply copies the entire **assets/rfo-basic** folder to the SD card.

Your application can use the **assets** folder, too. Most of the BASIC! file-handling commands look in the SD card file system first. If the file does not exist on the SD card, your program compiled into an APK looks in your projects's **res** folder, and finally in **assets**. Resources may be built into the APK in either the **res** or **assets** folder, but **res** is not treated as a file system and has more restrictive naming rules. For example, the name of a resource in **res** must not contain spaces or hyphens.

Both **Byte.open** and **Text.open** are able to open resources in your APK. However, **Zip.open** cannot open resources; it looks only in the SD card file system. If you want to read a ZIP that is in **assets**, you must first copy it from **assets** to the SD card. Then you can open the SD card file with **Zip.Open**.

File.Exists looks only for files on the SD card, but **File.Type** works with items in **assets** as well. You can use them together to determine where an item is. The other **File.*** commands work with either files or resources. **Font.Load** can load a resource if it does not find the font file on the SD card.

Files in **assets** are **read-only**. Your program can create and modify files on the SD card. It cannot create or modify files in **assets**. If you want BASIC! to copy files from **assets** to the SD card, list them the "load_file_names" tag of **setup.xml** as described above in "**Advanced Customization with setup.xml**".

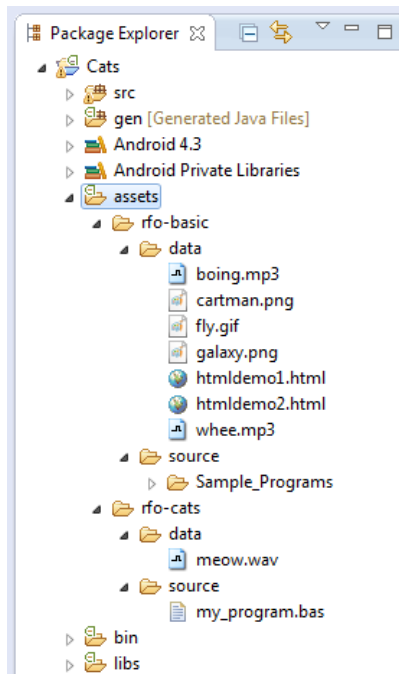
File names in **assets** are **case-sensitive**. If your program looks for a file on the SD card, the name is not case-sensitive: "meow.wav" and "Meow.WAV" are the same file. However, to find a file in **assets**, your program must name the file exactly as you put it in **assets**. **Audio.Load** aft, "meow.wav" will **not** find **assets/rfo-cats/data/Meow.WAV**.

For this practice APK, make the following changes:

1. In the **Package Explorer**, expand **Cats/assets/rfo-basic** and its **data** and **source** folders.
2. Right-click **assets** and select **New -> Folder**.
3. In the **Folder name:** field, enter "rfo-cats/data".
4. Click **Finish**.
5. In the same way, create "rfo-cats/source".
6. Drag **assets/rfo-basic/data/meow.wav** to **assets/rfo-cats/data/**.
7. Drag **assets/rfo-basic/source/my_program.bas** to **assets/rfo-cats/source/**.

Note: the top folder in **assets** must exactly match what you put in the "app_path" item in **res/values/setup.xml**. For this practice program, it is **rfo-cats**.

If you expand your new folders, your **Package Explorer** should look like this:



Your APK does not need anything in **assets/rfo-basic**. Delete the entire folder:

8. In the **Package Explorer**, right-click **Cats/assets/rfo-basic** and select **Delete**.
9. In the confirmation window that appears, click **OK**.
10. You are done making changes! It wouldn't hurt to do a **Project->Clean** here.

Testing the APK

We are now ready to test this practice APK.

The first thing you will do is to create a Keystore. The Keystore is used to sign your application. Google Play requires this signing. Android devices will not install unsigned APKs. You will use this one Keystore for all your APKs. Preserve and protect it. You will not be able to update your APK without it.

For more information about the Keystore and signing, see:

<http://developer.android.com/tools/publishing/app-signing.html>

1. In the **Package Explorer**, right-click **Cats**.
2. Select: **Android Tools -> Export Signed Application Package**.
3. In the **Project Checks** dialog box click **Next**.
4. Select **Create New Keystore**.
 - a. Provide a name and location for the Keystore.
 - b. Provide a password and confirm it.
 - c. Click **Next**.
5. Fill out the **Key Creation** dialog.
 - a. Pick any name for an Alias.

- b. Enter 25 for Validity (Years).
 - c. Click next.
6. In the **Destination and Key/Certificate Checks** dialog,
 - a. Browse to the folder where you want to put the APK.
 - b. Name the APK "Cats.apk".
 - c. Click next.
7. Now, install and run Cats.apk.

The APK will have the BASIC! icon. The name below the icon on your device will be "Cats!". Double click the icon to run your application.

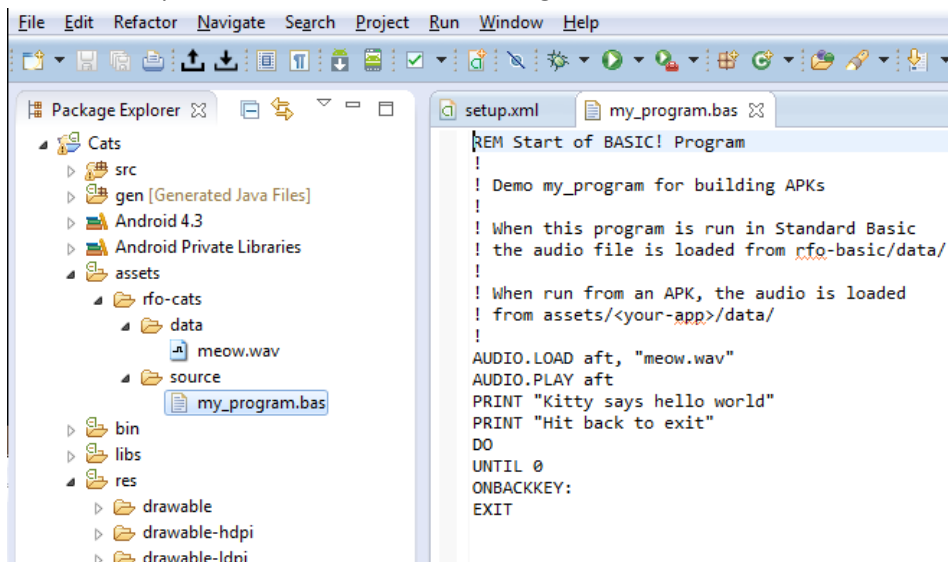
If you have reached this point successfully then you are ready to customize the APK for your application.

Start over with a new copy of Basic.zip but use names and information particular to your application and then continue below.

Installing a BASIC! Program into the Application

You must build your BASIC! program into your application by putting it into the **assets** folder, just as you saw in the practice program. One very simple way is to copy your text into **assets/<your-app>/my_program.bas**. **<your-app>** is the path you named in the **setup.xml** item "app_path".

1. In the Eclipse **Package Explorer**, expand the **assets** folder.
2. Double-click **assets/<your-app>/source/my_program.bas**.
3. The file will open in the edit window to the right:



Now you can edit the file directly in Eclipse. If you prefer, you can open your program outside of Eclipse, copy its contents to the clipboard, and paste it into the Eclipse editor. When you are finished, select **File->Save** from the menu bar, or just close the file by clicking the **X** on the file tab and **Yes** in the **Save Resource** dialog box.

For more complex projects, here is another way to do it.

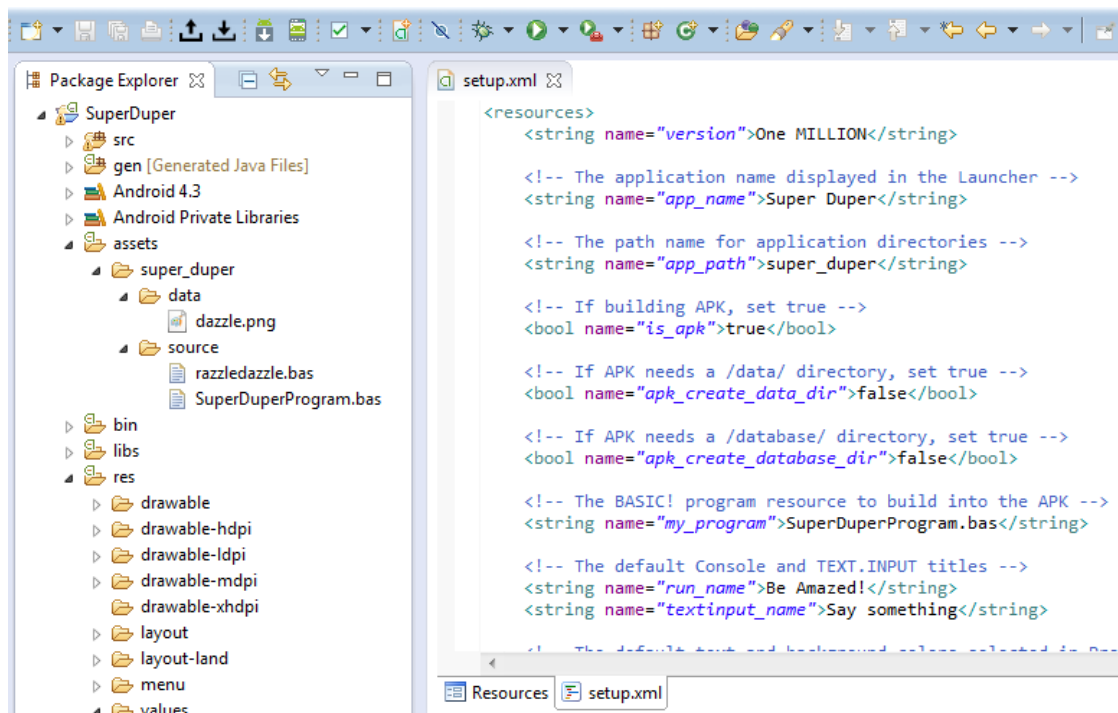
1. In the Eclipse **Package Explorer**, expand your program's **assets** folder.
2. If you have a **my_program.bas** in your **assets** folder, delete it.
3. In a file browser, browse to the file containing your program.
4. Drag your program from your file browser to your **assets/<your-app>/source** folder in Eclipse.
5. If your program uses **INCLUDE** files, drag them to the **source** folder, too.

With either method, you now have a program to run. If your program uses graphics, audio files, or other resources, you must put them in your Android **assets** folder, too.

If your program uses data files, drag them to your **assets/<your-app>/data** folder.

If your program uses data base files with the ".db" extension, create a new folder for them called **assets/<your-app>/databases** and drag your file into the new folder.

The result might look something like this:



Notice that the top folder in **assets** matches the "**app_path**" value, and there is a program in its **source** folder named the same as the "**my_program**" value.

Application Icons

Android specifies that Application icons must be provided in specific sizes: low dpi (36x36 pixels), medium dpi (48x48 pixels), high dpi (72x72), x-high dpi (96x96), and so on. The icons must be .png files. There are tens of thousands of free icons available on the web. Most image editing programs can be

used to re-sample the icons to the proper sizes and to save them as .png files. If you are not going to put your application on the Google Play Store then you do not really need to worry about getting this exactly right.

To get your icon into your application, in res, open drawable-ldpi, drawable-mdpi, drawable-hdpi.



For each of the icon sizes:

1. Outside of Eclipse, copy the icon file
2. In Eclipse, right click on the appropriate **drawable-** for the copied icon's size
3. Select **Paste**
4. Right click on the icon.png file and delete it
5. Select the newly pasted icon and rename it to "icon.png" by selecting **File -> Rename**.

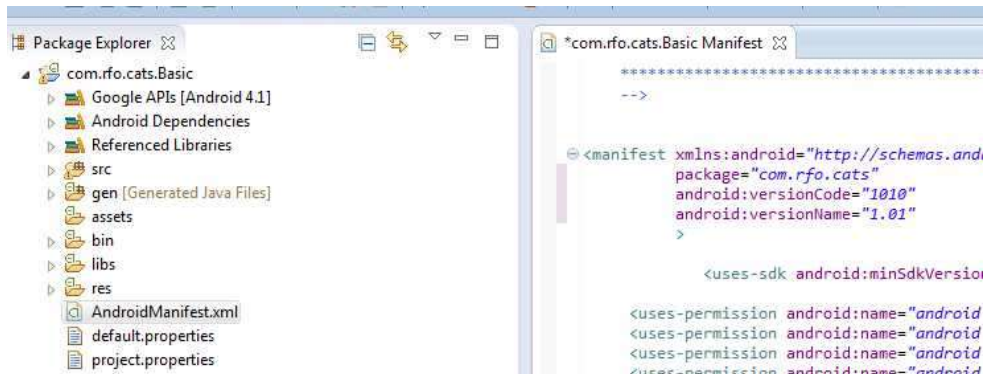
Yes, it is tedious work.

Modifications to the AndroidManifest.xml File

The **AndroidManifest.xml** file defines many aspects of how your application is built, and how it is presented to the Android system. For example, when you changed the package name, Eclipse wrote the change into the Android manifest.

You can build and run your application with no further changes to AndroidManifest.xml. However, the manifest carries information that your application may not need. You should customize it to suit your application.

In the **Package Explorer**, find the **AndroidManifest.xml** file. Double-click to open it:



Eclipse has special editors to display different views of the Manifest. If you don't see the actual XML text as shown here, click on the rightmost tab at the bottom of the window, labeled **AndroidManifest.xml**.

Setting the Version Number and Version Name

If you are going to put the application on the Google Play Store, you will need to change the version number and name for each new release.

Make the appropriate changes to the **android:versionCode** and **android:versionName** items of the **<manifest>** tag at the top of **AndroidManifest.xml**.

If you want to use the BASIC! **VERSION\$()** function to have your program read your version number, you will also have to change the version number in **res->values->strings**.

Permissions

BASIC! uses many features about which the APK user is warned and must approve. Your particular APK may not need all or any of these permissions. The permission notifications are contained in the **<uses-permission>** tags of the **AndroidManifest.xml**. They look like this:

```
<uses-permission android:name="android.permission. ... " />
```

Look them over. If you feel that your APK does not need them then delete or comment them out.

If your application uses the SD card, do not comment out:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Note: your application does **not** need **WRITE_EXTERNAL_STORAGE** permission if you use only the private storage named in the **SysPath** key of the **Program.info** command.

If you want your application to run when the device boots (see "Launch at device boot", below), do not comment out:

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
```

Be sure to test your APK after changing permissions.

Disable the Shortcut Launcher

Each Activity screen that BASIC! can present to a user is declared in an **<activity>** tag in the **AndroidManifest.xml** file. Your application may not use every Activity BASIC! offers. You may remove the unused **<activity>** tags if you wish, but they don't hurt anything if you leave them in.

However, the **<activity>** tags also tell the Android system what events should start your application. You probably do not want your application to respond to these two events:

- 1) A user can launch BASIC! by tapping a file with a ".bas" extension
- 2) A user can use the Shortcut Launcher to create a shortcut for a BASIC! program

You remove these by deleting their tags from the Manifest:

- 1) Find the **<activity>** tag that contains **android:name="Basic"**.
 - a) In it, find the **<intent-filter>** tag that contains **<data android:pathPattern=".*\\.bas" />**.
 - b) Remove all of the **<intent-filter>** tag, from **<intent-filter>** to **</intent-filter>**.
Do **not** remove the **<activity>** tag.



- 2) There are two shortcut tags:
 - a) Find and remove the **<activity>** tag that contains **android:name="LauncherShortcuts"**.
 - b) Find and remove the **<activity-alias>** tag that contains **android:name="CreateShortcuts"**.



Launch at device boot

Your APK can be set up to automatically launch just after the Android device has booted. This is accomplished by changing another item in **AndroidManifest.xml**.

Find the `<receiver>` tag with `android:name=".BootUpReceiver"`.

Change `android:enabled="false"` to `android:enabled="true"`.

You must also make sure your app has "RECEIVE_BOOT_COMPLETED" permission. See the "Permissions" section, above.

Preferences

There are certain preferences such as screen colors and font sizes that you have set for your application. The preferences that you will get with an APK will be BASIC!'s default preferences. You can change the default preferences if you wish.

Some preferences are simple check boxes. To change these, open the **res/xml** hierarchy and double click on the **settings.xml** file as shown below:

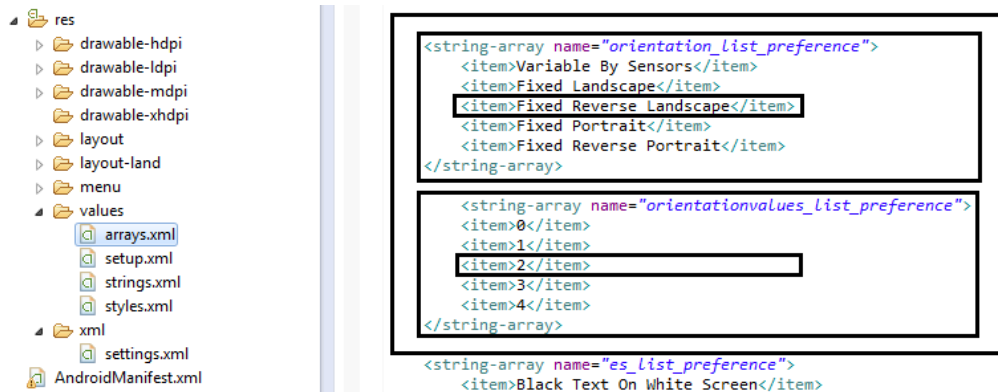
For example, to change the default Console Lines preference from lined console to unlined console, change the value on the indicated line from **"true"** to **"false"**.



If your app uses graphics mode, and you have determined it works correctly with hardware-accelerated graphics, you may want to change the default setting of the `CheckBoxPreference` named "gr_accel". The default is normally "false", but to enable graphics acceleration in an APK you must change it to "true".

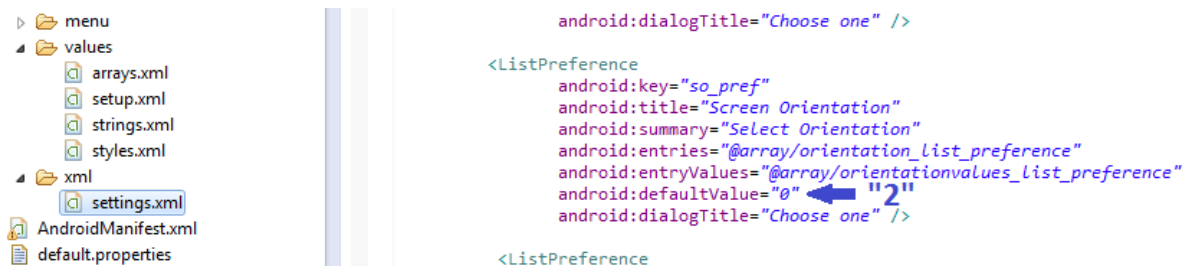
Other preferences are multiple-choice lists. To see the list values, open the **res/values** hierarchy and double click on **arrays.xml**. Each preference has two blocks. The top block lists the words that will be seen on the Android screen. The second block lists the internal names that correspond to the displayed words.

In the image below:



The section marked contains the names and values for the Screen Orientation preference. The top block is the display names. The bottom block is the internal values that correspond to the display name. For example the internal value of **Fixed Reverse Landscape** is **2**.

To set a default value for Screen Orientation, we need to go back to **settings.xml**:



Find the **<ListPreference>** tag with **android:title="Screen Orientation"**. The title is the preference name that you see on the Android screen. The default value is in the **android:defaultValue="0"** line. Here we see the default value for the screen orientation is "0". Looking at the Array.xml file we can see the "0" is the internal name for **Variable By Sensors**. To change the default value to **Fixed Reverse Landscape**, change the "0" to "2".

The other list preferences follow the same logic.

Note: Be sure to test your application with your chosen preferences before burning them into an APK.

Finished

Create your finished APK in the same way we created the practice APK.

Now that was not too bad, was it?

Appendix E – BASIC! Distribution License

BASIC! is distributed under the terms of the GNU General Public License which is reproduced here.

GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <<http://fsf.org/>>
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for
software and other kinds of works.

The licenses for most software and other practical works are designed
to take away your freedom to share and change the works. By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users. We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
any other work released this way by its authors. You can apply it to
your programs, too.

When we speak of free software, we are referring to freedom, not
price. Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you
these rights or asking you to surrender the rights. Therefore, you have
certain responsibilities if you distribute copies of the software, or if
you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether
gratis or for a fee, you must pass on to the recipients the same
freedoms that you received. You must make sure that they, too, receive
or can get the source code. And you must show them these terms so they
know their rights.

Developers that use the GNU GPL protect your rights with two steps:
(1) assert copyright on the software, and (2) offer you this License
giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains
that there is no warranty for this free software. For both users' and
authors' sake, the GPL requires that modified versions be marked as
changed, so that their problems will not be attributed erroneously to
authors of previous versions.

Some devices are designed to deny users access to install or run
modified versions of the software inside them, although the manufacturer

can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If

the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains

in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
- b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
- c) You must license the entire work, as a whole, under this

License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to

copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this

License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible

for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have

actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

Apache Commons

Portions of BASIC! use Apache Commons.

Apache Commons Net

Copyright 2001-2012 The Apache Software Foundation

This product includes software developed by

The Apache Software Foundation (<http://www.apache.org/>).