BASIC! Language Reference

Originally published as De_Re_Basic!

Original Author Paul Laughton, 2011

Edited by Robert A. Rioja

robrioja@gmail.com

http://www.RvAdList.com

Document Version 2023-09-05

Table of Contents

1 Introduction

1.1 BASIC!

BASIC! is an Android interpreter for the Basic language. It is also known as Basic!, rfo-basic, and rfo-basic!.

1.2 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 the BASIC! forum.

1.3 Credits

Thanks to Paul Laughton, the original creator of BASIC! and its original documentation. 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.

1.4 Documentation

This document, ***Basic! Language Reference***, was developed from the original ***De_Re_BASIC!*** document. It is a companion to the ***Basic! User Manual***, which was also developed from the original ***De_Re_BASIC!*** document.

"De Re" is Latin for "of the thing" or "about". Therefore, “De Re Basic!” means “About Basic!”.

The ***Basic! User Manual*** and the ***Basic! Language Reference*** were edited, and are maintained, by Robert A. Rioja.

1.4.1 To Do

Check links and add appendices as needed.

2 Basic Syntax

2.1 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!"
```

Note that 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.

2.2 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"
```

3 Android Device

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	89014103211118510720
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.

Information returned by **Device** is static. To get dynamic information, use the **Phone.Info** command.

3.1 Device

Syntax: Device <svar>

or

Syntax: Device <nexp>|<nvar>

The first form of this command 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. 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 element is one item, "<k> = <v>"
SPLIT lang_line$[], info$[6], " = "    % split the Language item, two-element
array
lang$ = lang_line$[2]                  % lang$ is language name, like "English"
```

The second form of this command returns information about your Android device in a Bundle pointed at by <nexp>|<nvar>. 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.

```
DEVICE info                     % all info in a Bundle
BUNDLE.GET info, "Language", lang$ % lang$ is the language name, like "English"
```

3.2 Device.language

Syntax: 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.

3.3 Device.locale

Syntax: 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.

3.4 Device.OS

Syntax: Device.OS <svar>

Returns the Android version string without needing the READ_PHONE_STATE permission.

This is a string, and it needs to be converted to a number if needed. Note that the string might be something like “4.0.3”, and may fail the Is_number or Val functions.

3.5 Phone.info

Syntax: 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**, **EDVOrevB**, **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.

3.6 Screen

Syntax: 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.

3.7 Screen.rotation

Syntax: 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.

3.8 Screen.size

Syntax: 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.

3.9 WiFi.info

Syntax: 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).

4 App Commands

4.1 App.broadcast

Syntax: 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".

4.2 App.start

Syntax: 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.

5 Array Commands

5.1 Array.average

Syntax: 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>.

5.2 Array.copy

Syntax: 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.

5.3 Array.delete

Syntax: Array.delete Array[]{, Array[]} ...

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

5.4 Array.dims

Syntax: 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.

5.5 Array.fill

Syntax: 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.

5.6 Array.length

Syntax: 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>.

5.7 Array.load

Syntax: 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"
```

5.8 Array.max

Syntax: 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>.

5.9 Array.min

Syntax: 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>.

5.10 Array.reverse

Syntax: 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]).

5.11 Array.search

Syntax: 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.

If only a segment of an array is used, the result will be relative to the start point.

Example:

```
Array.load ar[],10, 8,12, 20, 6, 7, 88, 11
Array.search ar[4, 3], 6, pos          % pos = 2 instead of 5
Array.search ar[4, 3], 88, pos         % pos=0, because 88 is
                                       % not part of the segment
```

5.12 Array.shuffle

Syntax: 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]).

5.13 Array.sort

Syntax: 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.

5.14 Array.std_dev

Syntax: 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>.

5.15 Array.sum

Syntax: 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>.

5.16 Array.variance

Syntax: 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>.

5.17 Dim

Syntax: 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]
```

5.18 UnDim

Syntax: UnDim Array[]{, Array[] } ...

“Undimensions” 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**.

6 Audio Commands

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.

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, shown in 50 Appendix - Supported media formats. 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+

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.

6.1 Audio.isdone

Syntax: 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
```

6.2 Audio.length

Syntax: 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>.

6.3 Audio.load

Syntax: 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"
```

and

```
   "../../Music/Blue Danube Waltz.mp3"
```

would access

```
   "<pref base drive>/Music/Blue Danube Waltz.mp3".
```

To load an internet stream, specify a full URL.

6.4 Audio.loop

Syntax: 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.

6.5 Audio.pause

Syntax: 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.

6.6 Audio.play

Syntax: 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
```

6.7 Audio.position.current

Syntax: Audio.position.current <nvar>

The current position in milliseconds of the currently playing file will be returned in <nvar>.

6.8 Audio.position.seek

Syntax: Audio.position.seek <nexp>

Moves the playing position of the currently playing file to <nexp> expressed in milliseconds.

6.9 Audio.record.start

Syntax: 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.

6.10 Audio.record.stop

Syntax: Audio.record.stop

Stops the previously started audio recording.

6.11 Audio.release

Syntax: 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.

6.12 Audio.stop

Syntax: Audio.stop

Audio.stop terminates the currently-playing music file. This command will be ignored is no file is playing. It is best to precede each **audio.play** command with an **audio.stop** command.

6.13 Audio.volume

Syntax: 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].

7 Background Commands and Functions

7.1 Background

Syntax: Background()

This 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.

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.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
```

7.2 Background.resume

Syntax: Background.resume

This command resumes an interrupted program. It should be included in an interrupt handler as described in section **OnBackground:**.

7.3 Home

Syntax: 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.

7.4 OnBackground:

Syntax: OnBackground:

Interrupt handler label to trap changes in the Background/Foreground state. The **Background()** function can be used to determine the new state. When done, execute the **Background.resume** command to resume the interrupted program.

7.5 WakeLock

Syntax: 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.

The following table lists the Code parameters.

Code	WakeLock Type	CPU	Screen	Keyboard Light	More Information
1	Partial WakeLock	On*	Off	Off	http://developer.android.com/reference/android/os/PowerManager.html#PARTIAL_WAKE_LOCK

2	Screen Dim	On	Dim	Off	http://developer.android.com/reference/android/os/PowerManager.html#SCREEN_DIM_WAKE_LOCK
3	Screen Bright	On	Bright	Off	http://developer.android.com/reference/android/os/PowerManager.html#SCREEN_BRIGHT_WAKE_LOCK
4	Full WakeLock	On	Bright	Bright	http://developer.android.com/reference/android/os/PowerManager.html#FULL_WAKE_LOCK
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	More Information
1	Acquire causes wakeup	Turn screen on when wakelock is acquired.	http://developer.android.com/reference/android/os/PowerManager.html#ACQUIRE_CAUSES_WAKEUP
2	On after release	Reset screen timeout when wakelock is released.	http://developer.android.com/reference/android/os/PowerManager.html#ON_AFTER_RELEASE
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.

7.6 WifiLock

Syntax: 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	More Information

1	Scan Only	WiFi kept active, but only operation is initiating scan and reporting scan results.	http://developer.android.com/reference/android/net/wifi/WifiManager.html#WIFI_MODE_SCAN_ONLY
2	Full	WiFi behaves normally.	http://developer.android.com/reference/android/net/wifi/WifiManager.html#WIFI_MODE_FULL
3	Full High Performance	WiFi operates at high performance with minimum packet loss and low latency*.	http://developer.android.com/reference/android/net/wifi/WifiManager.html#WIFI_MODE_FULL_HIGH_PERF
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.

8 Bluetooth Commands

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.

8.1 Bt.close

Syntax: Bt.close

Closes any previously opened Bluetooth connection. Bluetooth will automatically be closed when the program execution ends.

8.2 Bt.connect

Syntax: 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.

8.3 Bt.device.name

Syntax: 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.

8.4 Bt.disconnect

Syntax: 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.

8.5 Bt.onReadReady.resume

Resumes an interrupted program. See 8.13 OnBtReadReady: for more information.

8.6 Bt.open

Syntax: 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.

8.7 Bt.read.bytes

Syntax: 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.

8.8 Bt.read.ready

Syntax: 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.

8.9 Bt.reconnect

Syntax: 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**.

8.10 Bt.set.UUID

Syntax: 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.

8.11 Bt.status

Syntax: 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".

8.12 Bt.write

Syntax: 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.

8.13 OnBtReadReady:

Interrupt handler that traps the arrival of a message received on the Bluetooth channel. If a Bluetooth message is ready (**Bt.read.ready** would return a non-zero value) BASIC! executes the statements after the **OnBtReadReady:** label, where you can read and handle the message. When done, execute the **Bt.onReadReady.Resume** command to resume the interrupted program.

9 Bundle Commands

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 a 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.

9.1.1 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.

```
BUNDLE.PUT b,"key1", 1.2
% try to put a value in the bundle pointed to by b
BUNDLE.PUT 10, key2$, value2
% try to put a value in the 10th bundle created
BUNDLE.REMOVE c + d, key$[3],
% 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.

9.2 Bundle.clear

Syntax: 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.

9.3 Bundle.contain

Syntax: Bundle.contain <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.

9.4 Bundle.create

Syntax: Bundle.create <pointer_nvar>

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

Example:

```
BUNDLE.CREATE bptr
```

9.5 Bundle.get

Syntax: 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
```

9.6 Bundle.keys

Syntax: 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
```

9.7 Bundle.put

Syntax: 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
```

9.8 Bundle.remove

Syntax: 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.

9.9 Bundle.type

Syntax: 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
```

10 Clipboard Commands

10.1 Clipboard.get

Syntax: Clipboard.get <svar>

Copies the current contents of the clipboard into <svar>

10.2 Clipboard.put

Syntax: Clipboard.put <sexp>

Places <sexp> into the clipboard.

11 Communication: Email, Phone and Text Commands

11.1 Email.send

Syntax: 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.

11.2 MyPhoneNumber

Syntax: 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.

11.3 Phone.call

Syntax: 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.

11.4 Phone.dial

Syntax: 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.

11.5 Phone.rcv.init

Syntax: 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.

11.6 Phone.rcv.next

Syntax: 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.

11.7 Sms.send

Syntax: 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.

11.8 Sms.rcv.init

Syntax: Sms.rcv.init

Prepare to intercept received SMS using the sms.rcv.next command.

11.9 Sms.rcv.next

Syntax: 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
```

12 Console Input and Interaction Commands

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 **Text.input** command operates on larger blocks of text, and **TGet** simulates terminal I/O.

12.1 Dialog.message

Syntax: 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.

12.2 Dialog.select

Syntax: 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.

12.3 Input

Syntax: 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).

12.4 Inkey$

Syntax: 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.

12.5 Kb.hide

Syntax: 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.

12.6 Kb.resume

Syntax: Kb.resume

Returns from a keyboard interrupt routine at **OnKbChange:**.

12.7 Kb.show

Syntax: 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.

12.8 Kb.showing

Syntax: 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**.

12.9 Kb.toggle

Syntax: 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.

12.10 Key.resume

Syntax: Key.resume

Returns from a keyboard interrupt routine at **OnKeyPress:**.

12.11 OnKbChange:

Syntax: 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.

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.

12.12 OnKeyPress:

Syntax: OnKeyPress:

Interrupt handler that traps a tap on any key. When done, execute the **Key.resume** command to resume the interrupted program.

12.13 Popup

Syntax: 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.

12.14 Select

Syntax: 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).

12.15 Text.input

Syntax: 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
```

12.16 TGet

Syntax: 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.

13 Console Output Commands

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 abut 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.

13.1 Cls

Syntax: Cls

Clears the Output Console screen.

13.2 Console.front

Syntax: 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.

13.3 Console.line.count

Syntax: 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.

13.4 Console.line.text

Syntax: Console.line.text <line_nexp>, <text_svar>

The text of the specified line number of the Console is copied to the <text_svar>.

13.5 Console.line.touched

Syntax: 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.

13.6 Console.save

Syntax: Console.save <filename_sexp>

The current contents of the Console is saved to the text file specified by the filename string expression.

13.7 Console.title

Syntax: 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".

13.8 ConsoleTouch.resume

Syntax: ConsoleTouch.resume

Returns from an iterrupt routine at **OnConsoleTouch:**

13.9 Print

Syntax: Print {<exp> {,|;}} ... or ? {<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:

Print can operate on either strings or numbers. Sometimes it has to try both ways before it knows what to do. First it evaluates an expression as a number. If that fails, it will 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.

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)
```

13.10 OnConsoleTouch:

Syntax: OnConsoleTouch:

Interrupt handler that traps a tap on a line printed on the Output Console. 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. When done, execute the **ConsoleTouch.resume** command to resume the interrupted program.

This handler 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.

To detect screen touches while in graphics mode, use **OnGrTouch:**.

14 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**.

14.1 Debug.dump.array

Syntax: 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.

14.2 Debug.dump.bundle

Syntax: Debug.dump.bundle <bundlePtr_nexp>

Dumps the Bundle pointed to by the Bundle Pointer numeric expression.

14.3 Debug.dump.list

Syntax: Debug.dump.list <listPtr_nexp>

Dumps the List pointed to by the List Pointer numeric expression.

14.4 Debug.dump.scalars

Syntax: 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.

14.5 Debug.dump.stack

Syntax: Debug.dump.stack <stackPtr_nexp>

Dumps the Stack pointed to by the Stack Pointer numeric expression.

14.6 Debug.echo.off

Syntax: Debug.echo.off

Turns off the Echo mode. Same as **Echo.off**.

14.7 Debug.echo.on

Syntax: Debug.echo.on

Turns on Echo mode. Same as **Echo.off**. 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.

14.8 Debug.off

Syntax: 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.

14.9 Debug.on

Syntax: 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.

14.10 Debug.print

Syntax: Debug.print

This command is exactly the same as the **Print** command except that the print will occur only while in the debug mode.

14.11 Debug.show

Syntax: 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.

14.12 Debug.show.array

Syntax: 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.

14.13 Debug.show.bundle

Syntax: 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.

14.14 Debug.show.list

Syntax: 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.

14.15 Debug.show.program

Syntax: 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.

14.16 Debug.show.scalars

Syntax: 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.

14.17 Debug.show.stack

Syntax: 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.

14.18 Debug.show.watch

Syntax: 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.

14.19 Debug.watch

Syntax: 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.

14.20 Echo.off

Syntax: Echo.off

Turns off the Echo mode. Same as **Debug.echo.off**.

14.21 Echo.on

Syntax: Echo.on

Turns on Echo mode. Same as **Debug.echo.off**.

15 Files and Paths

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, the base directory is **<pref base drive>/rfo-basic**. All file paths are relative to a subdirectory of the base directory.

15.1 Paths Explained

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. 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. 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.

15.2 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:

- 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".

15.3 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>.

15.4 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. 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.

15.5 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.

15.6 Files and Resources

If your program is compiled into an APK, file handling can be a little more complicated. The APK has several internal directories, but the two of interest to us are the **assets** and **res** directories.

Standard BASIC! loads its sample programs and the data files they need from the **assets** directory of the APK. Android treats the **assets** directory like a file system. At startup, BASIC! simply copies the entire **assets/rfo-basic** directory to the SD card.

Your application can use the **assets** directory, 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' **res** directory, and finally in the **assets** directory. Resources may be built into the APK in either the **res** or **assets** directory, 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**.

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**.

16 File Commands

16.1 Dir

See **File.dir**.

16.2 File.delete

Syntax: 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, or it could not be deleted, 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/".

If <path_sexp> contains more than one directory level, as in `"dir1/dir2/file1"`, this command will try to delete only the last item. So in the above example, only "file1" would be deleted while the folders "dir1" and "dir2" will not be deleted.

The following examples assume that "blabla" is a file:

```
file.delete L, "blabla"      % deletes file "blabla" in directory
                             % <pref base drive>/rfo-basic/data/

file.delete L, "dir1/blabla" % deletes file "blabla" in directory
                             % <pref base drive>/rfo-basic/data/dir1/
```

The following examples assume that "blabla" is a directory:

```
file.delete L, "blabla"      % deletes directory "blabla" from directory
                             % <pref base drive>/rfo-basic/data/
                             % but only if "blabla" is empty.

file.delete L, "dir1/blabla" % deletes directory "blabla" from directory
                             % <pref base drive>/rfo-basic/data/dir1/
                             % but only if "blabla" is empty.
```

16.3 File.dir

Syntax: 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/". **Dir** is a valid alias for this command.

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, "".

16.4 File.exists

Syntax: 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/".

16.5 File.mkdir

Syntax: 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>. **Mkdir** is a valid alias for this command.

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".

If <path_sexp> contains more than one directory level, all levels will be created. For example:

```
file.makedir "one/two"
```

will create "<pref base drive>/rfo_basic/data/one/two".

16.6 File.rename

Syntax: 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. **Rename** is a valid alias for this command.

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.

16.7 File.root

Syntax: 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>.

16.8 File.size

Syntax: 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/".

16.9 File.type

Syntax: 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

16.10 GrabFile

Syntax: 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$
```

16.11 GrabURL

Syntax: 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.

16.12 Mkdir

See **File.mkdir**.

16.13 Rename

See **File.rename**.

17 File Byte I/O Commands

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.

17.1 Byte.close

Syntax: Byte.close <file_table_nexp>

Closes the previously opened file.

17.2 Byte.copy

Sytax: 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 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**.

17.3 Byte.eof

Syntax: 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.

17.4 Byte.open

Syntax: 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 opened for read does not exist then <file_table_nvar> will be set to -1. The program can check for this and either create the file or report the error to the user, possibly using the **GetError$()** function.

17.5 Byte.position.get

Syntax: 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.

17.6 Byte.position.mark

Syntax: 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 **Files and Paths → Mark and Mark Limit**.

17.7 Byte.position.set

Syntax: 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.

17.8 Byte.read.buffer

Syntax: 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.

17.9 Byte.read.byte

Syntax: 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 <= byte <= 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
```

17.10 Byte.read.number

Syntax: 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.

17.11 Byte.truncate

Syntax: 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.

17.12 Byte.write.buffer

Syntax: 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.writeln**, except that it will have no added newline.

17.13 Byte.write.byte

Syntax: 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.

17.14 Byte.write.number

Syntax: 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.

18 File Text I/O Commands

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/".

18.1 Text.close

Syntax: 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.

18.2 Text.eof

Syntax: 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.

18.3 Text.open

Syntax: 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.

18.4 Text.position.get

Syntax: 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.

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 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.

18.5 Text.position.mark

Syntax: 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 **Files and Paths → Mark and Mark Limit**.

18.6 Text.position.set

Syntax: 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)**.

18.7 Text.readln

Syntax: 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.

18.8 Text.writeln

Syntax: 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.

Text.writeln with no parameters writes a newline. 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.

19 File ZIP I/O Commands

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.

19.1 Zip.close

Syntax: Zip.close <file_table_nexp>

Closes the previously opened ZIP file.

19.2 Zip.count

Syntax: 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.

19.3 Zip.dir

Syntax: 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, "".

19.4 Zip.open

Syntax: Zip.open {r|w}, <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.

19.5 Zip.read

Syntax: 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**.

19.6 Zip.write

Syntax: Zip.write <file_table_nexp> ,<buffer_sexp>, <file_name_sexp>

Writes the 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**.

20 Font Commands

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.

20.1 Font.clear

Syntax: 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.

20.2 Font.delete

Syntax: 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.

20.3 Font.load

Syntax: 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.

21 FTP Client Commands

These FTP commands implement a FTP Client

21.1 Ftp.cd

Syntax: Ftp.cd <new_directory_sexp>

Changes the current working directory to the specified new directory.

21.2 Ftp.close

Syntax: Ftp.close

Disconnects from the FTP server.

21.3 Ftp.delete

Syntax: Ftp.delete <filename_sexp>

Deletes the specified file.

21.4 Ftp.dir

Syntax: 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
```

21.5 Ftp.get

Syntax: 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".

21.6 Ftp.mkdir

Syntax: Ftp.mkdir <directory_sexp>

Creates a new directory of the specified name.

21.7 Ftp.open

Syntax: 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"
```

21.8 Ftp.put

Syntax: 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.

21.9 Ftp.rename

Syntax: Ftp.rename <old_filename_sexp>, <new_filename_sexp>

Renames the specified old filename to the specified new file name.

21.10 Ftp.rmdir

Syntax: Ftp.rmdir <directory_sexp>

Removes (deletes) the specified directory if and only if that directory is empty.

22 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.

22.1 GPS Control Commands

22.1.1 Gps.close

Syntax: 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.

22.1.2 Gps.open

Syntax: 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.

22.1.3 Gps.status

Syntax: 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
```

22.2 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 accuracty 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.

22.2.1 Gps.accuracy

Syntax: 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.

22.2.2 Gps.altitude

Syntax: Gps.altitude <nvar>

Returns the altitude in meters in <nvar>.

22.2.3 Gps.bearing

Syntax: Gps.bearing <nvar>

Returns the bearing in compass degrees in <nvar>.

22.2.4 Gps.latitude

Syntax: Gps.latitude <nvar>

Returns the latitude in decimal degrees in <nvar>.

22.2.5 Gps.location

Syntax: 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**.

22.2.6 Gps.longitude

Syntax: Gps.longitude <nvar>

Returns the longitude in decimal degrees in <nvar>.

22.2.7 Gps.provider

Syntax: 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, "".

22.2.8 Gps.satellites

Syntax: 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.

22.2.9 Gps.speed

Syntax: Gps.speed <nvar>

Returns the speed in meters per second in <nvar>.

22.2.10 Gps.time

Syntax: Gps.time <nvar>

Returns the time of the last GPS fix in milliseconds since "the epoch", January 1, 1970, UTC.

23 Graphics

23.1 Introduction

23.1.1 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.

23.1.2 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**.

23.1.3 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.

23.1.4 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.

23.1.5 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.

23.1.6 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.

23.1.6.1 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.

23.1.6.2 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.

23.1.7 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.

23.1.7.1 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-coodinate **left** are colored. Pixels with y-coordinate **top** are colored.

right and **bottom** values are "exclusive": pixels with x-coodinate **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.

23.1.7.2 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.

23.1.7.3 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, and the object is filled in as described in FILL. Because both parts are the same color, and there are never uncolored pixels beween 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.

23.1.8 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.

23.2 Graphics Setup Commands

23.2.1 Gr.brightness

Syntax: 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).

23.2.2 Gr.close

Syntax: 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.

23.2.3 Gr.cls

Syntax: 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.

23.2.4 Gr.color

Syntax: 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*" 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. 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
```

23.2.5 Gr.front

Syntax: 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.

23.2.6 Gr.open

Syntax: 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.

23.2.7 Gr.orientation

Syntax: 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.

23.2.8 Gr.render

Syntax: 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.

23.2.9 Gr.scale

Syntax: 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.

23.2.10 Gr.screen

Syntax: 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.

23.2.11 Gr.set.antialias

Syntax: 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.

23.2.12 Gr.set.stroke

Syntax: 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.

23.2.13 Gr.statusbar

Syntax: 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 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.

23.2.14 Gr.statusbar.show

Syntax: 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**.

23.3 Graphics 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**.

23.3.1 Gr.arc

Syntax: 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).

23.3.2 Gr.circle

Syntax: 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".

23.3.3 Gr.line

Syntax: 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".

23.3.4 Gr.oval

Syntax: 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".

23.3.5 Gr.point

Syntax: 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".

23.3.6 Gr.poly

Syntax: 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**.

23.3.7 Gr.rect

Syntax: Gr.rect <ob_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".

23.3.8 Gr.set.pixels

Syntax: 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 in 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.

23.4 Graphics 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
```

23.4.1 Gr.group

Syntax: 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".

23.4.2 Gr.group.getDL

Syntax: 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".

23.4.3 Gr.group.list

Syntax: 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".

23.4.4 Gr.group.newDL

Syntax: 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".

23.5 Graphics Hide and Show Commands

23.5.1 Gr.hide

Syntax: 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.

23.5.2 Gr.show

Syntax: 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.

23.5.3 Gr.show.toggle

Syntax: 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.

23.6 Graphics 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.

23.6.1 Gr.bounded.touch

Syntax: 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.

23.6.2 Gr.bounded.touch2

Syntax: 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.

23.6.3 Gr.onGrTouch.resume

Syntax: Gr.onGrTouch.resume

This command resumes an interrupted program. It should be included in an interrupt handler as described in section **OnGrTouch:**.

23.6.4 Gr.touch

Syntax: 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
```

23.6.5 Gr.touch2

Syntqax: Gr.touch2 touched, x, y

The same as **Gr.touch** except that it reports on second simultaneous touch of the screen.

23.6.6 OnGrTouch:

Syntax: OnGrTouch:

Interrupt handler that traps any touch on the Graphics screen. When done, execute the **Gr.onGrTouch.resume** command to resume the interrupted program.

To detect touches on the Output Console (not in Graphics mode), use **OnConsoleTouch:**.

23.7 Graphics Text Commands

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*".

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.

23.7.1 Gr.get.textbounds

Syntax: 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** 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
```

23.7.2 Gr.text.align

Syntax: 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.

23.7.3 Gr.text.bold

Syntax: 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.

23.7.4 Gr.text.draw

Syntax: 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.

23.7.5 Gr.text.height

Syntax: 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)
```

23.7.6 Gr.text.setfont

Syntax: 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-senstive: "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.

23.7.7 Gr.text.size

Syntax: 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.

23.7.8 Gr.text.skew

Syntax: 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.

23.7.9 Gr.text.strike

Syntax: 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.

23.7.10 Gr.text.typeface

Syntax: 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.

23.7.11 Gr.text.underline

Syntax: 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.

23.7.12 Gr.text.width

Syntax: 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.**

23.8 Graphics Bitmap Commands

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..

23.8.1 Gr.bitmap.create

Syntax: 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.

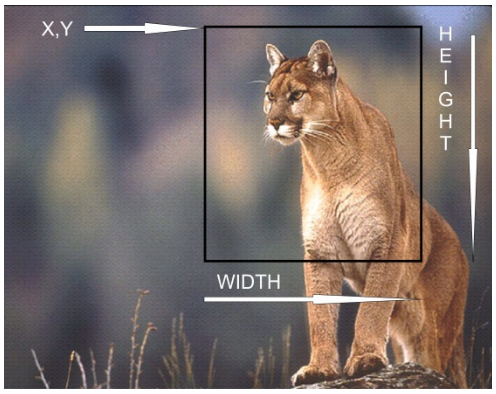
Returns a pointer to the created bitmap in the <bitmap_ptr_nvar> variable for use with the other **Gr.bitmap** commands. If there is not enough memory available to create the bitmap, the returned bitmap pointer is -1. Call **GetError$()** for information about the failure.

23.8.2 Gr.bitmap.crop

Syntax: 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_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.



23.8.3 Gr.bitmap.delete

Syntax: 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.

23.8.4 Gr.bitmap.draw

Syntax: 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".

23.8.5 Gr.bitmap.drawinto.end

Syntax: 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 **Gr.bitmap.draw** command for that bitmap.

23.8.6 Gr.bitmap.drawinto.start

Syntax: 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.

23.8.7 Gr.bitmap.fill

Syntax: 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.

23.8.8 Gr.bitmap.load

Syntax: 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.

23.8.9 Gr.bitmap.save

Syntax: 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 range for <quality_nexp> is 0 to 100. The default is 50.

The file will be saved as a PNG file if the filename ends in anything else (including ".png").

23.8.10 Gr.bitmap.scale

Syntax: 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.

23.8.11 Gr.bitmap.size

Syntax: 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.

23.8.12 Gr.get.bmpixel

Syntax: 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.

23.9 Graphics Paint Commands

23.9.1 Gr.paint.copy

Syntax: 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
```

23.9.2 Gr.paint.get

Syntax: 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.

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.

23.9.3 Gr.paint.reset

Syntax: 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.

23.10 Graphics 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.

23.10.1 Gr.rotate.end

Syntax: 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.NewDl** then you will need to include the **Gr.rotate.start** and **Gr.rotate.end** objects.

23.10.2 Gr.rotate.start

Syntax: 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.

23.11 Graphics 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 many 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.

23.11.1 Gr.camera.autoshoot

Syntax: 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".

23.11.2 Gr.camera.manualShoot

Syntax: 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.

23.11.3 Gr.camera.select

Syntax: 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.

23.11.4 Gr.camera.shoot

Syntax: 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."

23.12 Graphics Miscellaneous Commands

23.12.1 Gr.clip

Syntax: Gr.clip <object_ptr_nexp>, <left_nexp>, <top_nexp>, <right_nexp>, <bottom_nexp>{, <RO_nexp>}

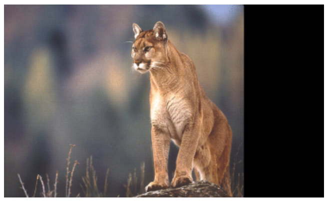
Objects that are drawn after this command is issued will be drawn only within the bounds (clipped) of the clip rectangle specified by the "left, top, right, bottom" numeric expressions.

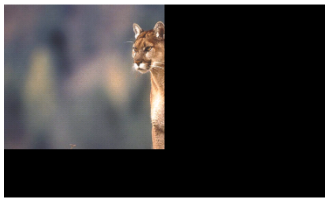
The final parameter is the Region Operator, <RO_nexp>. The Region Operator prescribes how this clip will interact with everything else you are drawing on the screen or bitmap. If you issue more than one **Gr.clip** command, the RO prescribes the interaction between the current **Gr.clip** rectangle and the previous one. The RO values are:

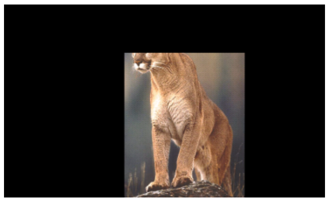
0	**Intersect**
1	**Difference**
2	**Replace**
3	**Reverse Difference**
4	**Union**
5	**XOR**

The Region Operator parameter is optional. If it is omitted, the default action is **Intersect**.

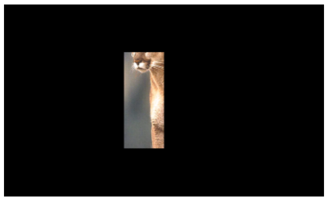
Examples:

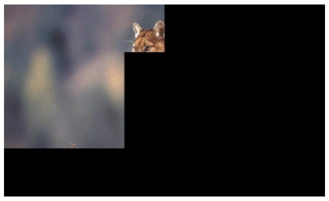

Original

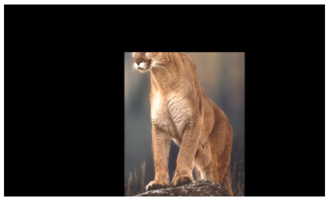

Clip 1

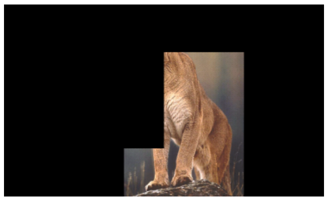

Clip 2

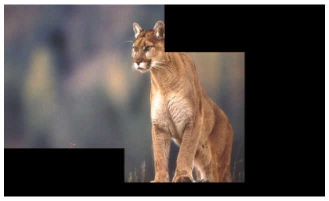
Clip 2 applied to Clip 1 with RO parameter on Clip 2

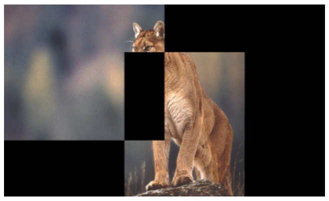

0 = Intersect


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.

23.12.2 Gr.getDL

Syntax: 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).

23.12.3 Gr.get.params

Syntax: 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**.

23.12.4 Gr.get.pixel

Syntax: 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.

23.12.5 Gr.get.position

Syntax: 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.

23.12.6 Gr.get.type

Syntax: 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.

23.12.7 Gr.get.value

Syntax: 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**.

23.12.8 Gr.modify

Syntax: 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
	text	x	y				text	paint	alpha
MODI-FIERS	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.

23.12.8.1 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
```

23.12.9 Gr.move

Syntax: 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.

23.12.10 Gr.newDL

Syntax: 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.

23.12.11 Gr.save

Syntax: 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.

23.12.12 Gr.screen.to_bitmap

Syntax: 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.

23.12.13 Gr_collision

Syntax: 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.

24 HTML Commands

HTML commands allow you to interact with the Android Webview control.

24.1 Browse

Syntax: Browse <url_sexp>

If <url_sexp> starts with "http…" then the internet site specified by <url_sexp> will be opened and displayed.

24.2 Html.clear.cache

Syntax: Html.clear.cache

Clears the HTML cache.

24.3 Html.clear.history

Syntax: Html.clear.history

Clears the HTML history.

24.4 Html.close

Syntax: Html.close

Closes the HTML engine and display.

24.5 Html.get.datalink

Syntax: 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>
```

24.6 Html.go.back

Syntax: Html.go.back

Go back one HTML screen, if possible.

24.7 Html.go.forward

Syntax: Html.go.forward

Go forward one HTML screen, if possible.

24.8 Html.load.string

Syntax: 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/
```

24.9 Html.load.url

Syntax: 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.

This command can also be used to execute JavaScripts that were included in the HTML file that is currently displayed. For example, the command:

```
HTML.LOAD.URL "javascript:myFunction();"
```

will call the JavaScript function myFunction(). The function must already exist in the HTML that was previously loaded via HTML.LOAD.STRING or HTML.LOAD.URL. For example:

```
<html>
<head>
 <script>
  function myFunction() {
   ...
  }
 </script>
</head>
...
</html>
```

If you need to pass parameters to the function, they must be included as strings:

Basic:

```
HTML.LOAD.URL "javascript:myFunction('123');"
```

HTML:

```
<html>
<head>
 <script>
  function myFunction(Parm) {
   ...
  }
 </script>
</head>
...
</html>
```

Note that JavaScript will accept single-quotes (') as well as double-quotes (").

If you need to pass parameters that are already in Basic variables you have to include them as part of the string:

Basic:

```
Alfa = 123
Beta$ = "abc"

HTML.LOAD.URL "javascript:myFunction('" + str$(Alfa) + ~
              "', '" + Beta$ + "');"
```

HTML:

```
<html>
<head>
 <script>
  function myFunction(Parm1, Parm2) {
   ...
  }
  </script>
</head>
...
</html>
```

Note that numeric variables must be converted to strings to be passed as part of one big string. In the above example, the string will become:

```
"javascript:myFunction('123', 'abc');"
```

24.10 Html.open

Syntax: 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.

24.11 Html.orientation

Syntax: 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.

24.12 Html.post

Syntax: 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.

24.13 Http.post

Syntax: 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.

25 Interrupts, Event Handlers and Errors

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.

25.1 Interrupt Labels

BASIC! supports trapping of the following events:

Interrupt Label	**Resume Statement**	**See Section**
OnBackground:	Background.resume	7.4 OnBackground:, 7.5 WakeLock
OnBackKey:	Back.resume	25.4 OnBackKey:, Back.resume
OnBtReadReady:	Bt.onReadReady.resume	8.13 OnBtReadReady:, 8.5 Bt.onReadReady.resume
OnConsoleTouch:	ConsoleTouch.resume	13.10 OnConsoleTouch:, 13.8 ConsoleTouch.resume
OnError:	None	25.2 OnError:
OnGrTouch:	Gr.onGrTouch.resume	23.6.6 OnGrTouch:, 23.6.3 Gr.onGrTouch.resume
OnKbChange:	Kb.resume	12.11 OnKbChange:, 12.10 Key.resume
OnKeyPress:	Key.resume	12.12 OnKeyPress:, 12.10 Key.resume
OnLowMemory:	LowMemory.resume	25.5 OnLowMemory:, LowMemory.resume
OnMenuKey:	MenuKey.resume	25.6 OnMenuKey:, MenuKey.resume
OnTimer:	Timer.resume	48.1 OnTimer:, 48.3 Timer.resume

Most of the above interrupt labels and their Resume commands are described in various sections of this manual. Some do not have appropriate sections, and are described here.

25.2 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"). 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.

25.3 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$()** to retrieve error information. For example, if **Text.open** cannot open a file, it sets the file pointer to -1, and writes a **GetError$()** message such as "<filename> not found". If no error occurred, **GetError$()** returns "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.

25.4 OnBackKey:, Back.resume

Interrupt handler 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).

25.5 OnLowMemory:, LowMemory.resume

Interrupt handler 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.

25.6 OnMenuKey:, MenuKey.resume

Interrupt handler 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.

26 List Commands

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**.

26.1 List.add

Syntax: 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"
```

26.2 List.add.list

Syntax: 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).

26.3 List.add.array

Syntax: 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).

26.4 List.clear

Syntax: List.clear <pointer_nexp>

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

26.5 List.create

Syntax: 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.

26.6 List.get

Syntax: 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).

26.7 List.insert

Syntax: 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).

26.8 List.remove

Syntax: 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.

26.9 List.replace

Syntax: 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.

26.10 List.search

Syntax: 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 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.

26.11 List.size

Syntax: List.size <pointer_nexp>, <nvar>

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

26.12 List.toArray

Syntax: 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.

26.13 List.type

Syntax: 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.

27 Math Functions

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

27.1 Abs

Syntax: Abs(<nexp>)

Returns the absolute value of <nexp>.

27.2 Acos

Syntax: Acos(<nexp>)

Returns the arc cosine of the angle <nexp>, in the range of 0.0 through pi.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.

27.3 Asin

Syntax: 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.

27.4 Atan

Syntax: 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.

27.5 Atan2

Syntax: 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.

27.6 Band

Syntax: 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
```

27.7 Bnot

Syntax: 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 ffffffffffffedcb
```

27.8 Bor

Syntax: 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
```

27.9 Bxor

Syntax: 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
```

27.10 Cbrt

Syntax: Cbrt(<nexp>)

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

27.11 Ceil

Syntax: Ceil(<nexp>)

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

27.12 Cos

Syntax: Cos(<nexp>)

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

27.13 Cosh

Syntax: Cosh(<nexp>)

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

27.14 ExpXP

Syntax: ExpXP(<nexp>)

Returns e raised to the <nexp> power.

27.15 Floor

Syntax: Floor(<nexp>)

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

27.16 Frac

Syntax: 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)**".

27.17 Hypot

Syntax: Hypot(<nexp_x>, <nexp_y)

Returns Sqr(x^2+y^2) without intermediate overflow or underflow.

27.18 Int

Syntax: 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.

27.19 Log

Syntax: Log(<nexp>)

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

27.20 Log10

Syntax: Log10(<nexp>)

Returns the base 10 logarithm of the <nexp>.

27.21 Max

Syntax: Max(<nexp>, <nexp>)

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

27.22 Min

Syntax: Min(<nexp>, <nexp>)

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

27.23 Mod

Syntax: Mod(<nexp1>, <nexp2>)

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

27.24 Pi

Syntax: Pi()

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

27.25 Pow

Syntax: Pow(<nexp1>, <nexp2>)

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

27.26 Round

Syntax: Round(<value_nexp>{, <count_nexp>{, <mode_sexp>}})

In it 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 ±∞)

"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")**.

27.27 Sgn

Syntax: 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

27.28 Shift

Syntax: 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.

27.29 Sin

Syntax: Sin(<nexp>)

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

27.30 Sinh

Syntax: Sinh(<nexp>)

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

27.31 Sqr

Syntax: 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.

27.32 Tan

Syntax: Tan(<nexp>)

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

27.33 ToDegrees

Syntax: ToDegrees(<nexp>)

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

27.34 ToRadians

Syntax: ToRadians(<nexp>)

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

28 Miscellaneous Commands

28.1 Headset

Syntax: 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.

28.2 Notify

Syntax: Notify <title_sexp>, <subtitle_sexp>, <alert_sexp>, <wait_lexp>

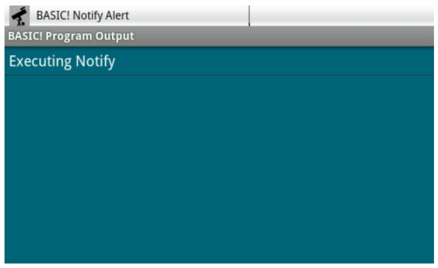
This command will cause a Notify object to be placed in the Notify (Status) bar. The Notify object displays the BASIC! app icon and the <alert_sexp> text. The user taps the Notify object to open the notification window. Your program's notification displays the <title_sexp> and <subtitle_sexp> text.

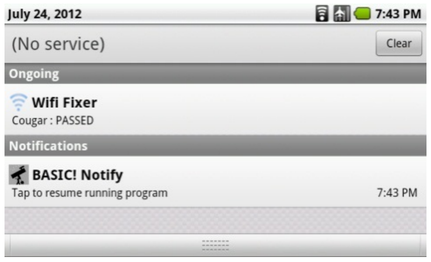
The code snippet and screenshots shown below demonstrate the placement of the parameter strings.

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"
```


BASIC! Notify Alert
BASIC! Program Output
Executing Notify


July 24, 2012 7:43 PM
(No service) Clear
Ongoing
Wifi Fixer
Cougar : PASSED
Notifications
BASIC! Notify
Tap to resume running program 7:43 PM

Note: the icon that appears in the Notify object will be the icon for the application in user-built apk.

28.3 Pause

Syntax: 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
```

28.4 Swap

Syntax: Swap <nvar_a>|<svar_a>, <nvar_b>|<svar_b>

The values and in "a" and "b" numeric or string variables are swapped. The two variables must be of the same type.

28.5 Tone

Syntax: <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.

28.6 Vibrate

Syntax: 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**.

28.7 Volume Keys

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.

Volume Keys		
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

28.7.1 VolKeys.off

Syntax: VolKeys.off

Disables the usual action of the keys listed in the **Volume Keys** table. Your program can still detect these keypresses, but BASIC! does not pass the events on to the Android system.

28.7.2 VolKeys.on

Syntax: VolKeys.on

Enables the usual action of the keys listed in the **Volume Keys** table. This is the default setting when your BASIC! program starts.

29 Program Control, Execution and Status Commands

29.1 Include

Syntax: 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, 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 the above example, the contents of the included file will 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.

29.2 Program.info

Syntax: 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	Numer (Logical)	Returns 1.0 (true) if the current program is being run from a standalone user-built APK. Returns 0.0 (false) if the program is being from from the BASIC! Editor or a Launcher Shortcut.

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

SysPath is of particular interest to you if you build a BASIC! program as an application in a standalone apk. 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.

29.3 Run

Syntax: 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 a 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.

29.4 Version$

Syntax: Version$()

Returns the version number of BASIC! as a string.

30 Program Flow Statements

30.1 Do / Until

Syntax: 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".

30.1.1 D_U.continue

Syntax: 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.

30.1.2 D_U.break

Syntax: 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.

30.2 For - To - Step / Next

Syntax: 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".

30.2.1 F_N.continue

Syntax: 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.

30.2.2 F_N.break

Syntax: 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.

30.3 If / Then / Else / Elseif / Endif

Syntax: 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.

30.4 If / Then / Else

Syntax: 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**.

30.5 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.

30.5.1 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.

30.5.2 Sw.begin

Syntax: 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 to 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 *first* 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.

30.5.3 Sw.case

Syntax: Sw.case <exp>, ...

or

Syntax: Sw.case <op><exp>

The first 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.

The 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.

30.5.4 Sw.break

Syntax: 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.case**s will be executed until a **Sw.break** or **Sw.end** is encountered.

30.5.5 Sw.default

Syntax: 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.

30.5.6 Sw.end

Syntax: Sw.end

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

30.6 While / Repeat

Syntax: 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.

30.6.1 W_R.continue

Syntax: 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".

30.6.2 W_R.break

Syntax: 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.

30.7 Labels, GoTo, GoSub, and Return

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.
```

30.7.1 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
```

30.7.2 GoTo

Syntax: GoTo <label>

or

Syntax: GoTo <index_nexp>, <label>...

For the first form of the **GoTo** statement, the next statement to be executed is the statement following <label>.

The second form is called 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**.

Example:

```
d = Floor(6 * Rnd() + 1)        % roll a six-sided die
GoTo d, Side1, Side2, Side3, Side4, Side5, Side6
Print "Welcome back!"
End

Side1:
  <statements>
.
.
.
Side6:
  <statements>
```

30.7.3 GoSub / Return

Syntax: GoSub <label> / Return

or

Syntax: GoSub <index_nexp>, <label>...

For the first form of the **GoSub** statement, the next statement to be executed is 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
```

The second form is called 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
```

30.8 End

Syntax: 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.

30.9 Exit

Syntax: Exit

Causes BASIC! to stop running and exit to the Android home screen.

31 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.

32 Random Number Generator

32.1 Randomize

Syntax: 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.

32.2 Rnd

Syntax: 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 <= n < 1).

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

33 Read Commands

33.1 Read.data

Syntax: 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.

33.2 Read.from

Syntax: 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.

33.3 Read.next

Syntax: 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.

34 Ringer Commands

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.

34.1 Ringer.get.mode

Syntax: Ringer.get.mode <nvar>

Returns the current ringtone mode in the numeric variable.

34.2 Ringer.get.volume

Syntax: Ringer.get.volume <vol_nvar> { , <max_nvar> }

Returns the ringer volume level in the numeric variable. Returns the maximum volume settting in <max_nvar>, if <max_nvar> is present.

34.3 Ringer.set.mode

Syntax: 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.

34.4 Ringer.set.volume

Syntax: 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.

35 Sensors

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 (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.

35.1 Sensors.close

Syntax: 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.

35.2 Sensors.list

Syntax: 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
```

35.3 Sensors.open

Syntax: 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.

35.4 Sensors.read

Syntax: 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.

36 Socket (TCP/IP) Commands

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 to 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.

36.1 Client Socket (TCP/IP) Commands

36.1.1 Socket.client.close

Syntax: Socket.client.close

Closes an open client side connection.

36.1.2 Socket.client.connect

Syntax: 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.client.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.

36.1.3 Socket.client.read.file

Syntax: 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
```

36.1.4 Socket.client.read.line

Syntax: 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.

36.1.5 Socket.client.read.ready

Syntax: 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.

36.1.6 Socket.client.server.ip

Syntax: Socket.client.server.ip <svar>

Return the IP of the server that this client is connected to in the string variable.

36.1.7 Socket.client.status

Syntax: 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

36.1.8 Socket.client.write.bytes

Syntax: 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.

36.1.9 Socket.client.write.file

Syntax: 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
```

36.1.10 Socket.client.write.line

Syntax: 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.

36.2 Server Socket (TCP/IP) Commands

36.2.1 Socket.myIP

Syntax: 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"
```

36.2.2 Socket.myIP

Syntax: 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.

36.2.3 Socket.server.client.ip

Syntax: Socket.server.client.ip <nvar>

Return the IP of the Client currently connected to the Server.

36.2.4 Socket.server.close

Syntax: Socket.server.close

Close the previously created Server. Any currently connected client will be disconnected.

36.2.5 Socket.server.connect

Syntax: 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.

36.2.6 Socket.server.create

Syntax: Socket.server.create <port_nexp>

Establish a Server that will listen to the Port specified by the numeric expression, <port_nexp>.

36.2.7 Socket.server.disconnect

Syntax: 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.

36.2.8 Socket.server.read.file

Syntax: 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
```

36.2.9 Socket.server.read.line

Syntax: 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.

36.2.10 Socket.server.read.ready

Syntax: 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 returns a non-zero value.

36.2.11 Socket.server.status

Syntax: 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

36.2.12 Socket.server.write.bytes

Syntax: 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.

36.2.13 Socket.server.write.file

Syntax: 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
```

36.2.14 Socket.server.write.line

Syntax: 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.

37 SoundPool Commands

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 my 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.

37.1 SoundPool.load

Syntax: 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.

37.2 SoundPool.open

Syntax: 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.

37.3 SoundPool.pause

Syntax: SoundPool.pause <streamID_nexp>

Pauses the playing of the specified stream. If the stream ID is zero, all streams will be paused.

37.4 SoundPool.play

Syntax: 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.

37.5 SoundPool.release

Syntax: SoundPool.release

Closes the SoundPool and releases all resources. **Soundpool.open** can be called to open a new SoundPool.

37.6 SoundPool.resume

Syntax: SoundPool.resume <streamID_nexp>

Resumes the playing of the specified stream. If the stream ID is zero, all streams will be resumed.

37.7 SoundPool.setPriority

Syntax: SoundPool.setPriority <streamID_nexp>, <priority_nexp>

Changes the priority of a playing stream.

The lowest priority is zero.

37.8 SoundPool.setRate

Syntax: 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.

37.9 SoundPool.setVolume

Syntax: 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.

37.10 SoundPool.stop

Syntax: SoundPool.stop <streamID_nexp>

Stops the playing of the specified stream.

37.11 SoundPool.unload

Syntax: SoundPool.unload <soundID_nexp>

The specified loaded sound is unloaded.

38 Speech Conversion

38.1 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.

38.1.1 TTS.init

Syntax: TTS.init

This command must be executed before speaking.

38.1.2 TTS.speak

Syntax: 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 **TTSts.speak** to finish, even if the "wait" flag was false.

38.1.3 TTS.speak.toFile

Syntax: 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.

38.1.4 TTS.stop

Syntax: 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.

38.2 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.

38.2.1 STT.listen

Syntax: 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 appers.

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.

38.2.2 STT.results

Syntax: 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.

38.2.2.1 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
```

38.2.2.2 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
```

38.2.2.3 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.

39 Sql Commands

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/).

An excellent online tutorial on SQL can be found at www.w3schools.com (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/ ".

39.1 Sql.close

Syntax: 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.

39.2 Sql.delete

Syntax: 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.

39.3 Sql.drop_table

Syntax: 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.

39.4 Sql.exec

Syntax: Sql.exec <DB_pointer_nvar>, <command_sexp>

Execute ANY non-query SQL command string ("CREATE TABLE", "DELETE", "INSERT", etc.) using a previously opened database.

39.5 Sql.insert

Syntax: 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.)

39.6 Sql.new_table

Syntax: 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.

39.7 Sql.next

Syntax: 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>}}
```

39.8 Sql.open

Syntax: 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 the same time. Each opened database must have its own, distinct pointer.

39.9 Sql.query

Syntax: 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. This will sort upper and lower case separately (case sensitive). For case insensitive sorts, replace the "ASC" or "DESC" (ascending or descending) part of the <order_sexp> argument with "COLLATE NOCASE ASC" OR "COLLATE NOCASE DESC" respectively.

If the Order parameter is omitted, the rows are not sorted. If the Order parameter is present, the Where parameter must also be present. If you want to return all rows, just set Where$ = "".

39.10 Sql.query.length

Syntax: 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.

39.11 Sql.query.position

Syntax: 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.

39.12 Sql.raw_query

Syntax: 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.

39.13 Sql.update

Syntax: 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.

40 Stack Commands

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.

40.1 Stack.clear

Syntax: Stack.clear <ptr_nexp>

The stack designated by <ptr_nexp> will be cleared.

40.2 Stack.create

Syntax: 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>.

40.3 Stack.isEmpty

Syntax: 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.

40.4 Stack.peek

Syntax: 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.

40.5 Stack.pop

Syntax: 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.

40.6 Stack.push

Syntax: 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.

40.7 Stack.type

Syntax: 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.

41 String Functions That Return a String

41.1 Bin$

Syntax: Bin$(<nexp>)

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

41.2 Chr$

Syntax: 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
```

41.3 Decode$

Syntax: Decode$(<type_sexp>, {<qualifier_sexp>}, <source_sexp>)

or

Syntax: Decode$(<charset_sexp>, <buffer_sexp>)

The first form of this command 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 "PBEWithMD5AndDES". 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. See 51 Appendix - urlencoded.
"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 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$()**, for a partial list of valid charsets.

Syntax: Decode$(<charset_sexp>, <buffer_sexp>)

The second form of this command 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$()**.

41.4 Encode$

Syntax: Encode$(<type_sexp>, {<qualifier_sexp>}, <source_sexp>)

or

Syntax: Encode$(<charset_sexp>, <source_sexp>)

The first form of this command 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 "PBEWithMD5AndDES". 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 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$()**, for a partial list of valid charsets.

Syntax: Encode$(<charset_sexp>, <source_sexp>)

The second form of this command 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$()**.

41.5 Format$

Syntax: 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 ouput. 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.

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

41.6 Format_using$

Syntax: 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)
```

41.7 Hex$

Syntax: Hex$(<nexp>)

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

41.8 Int$

Syntax: Int$(<nexp>)

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

41.9 Left$

Syntax: Left$(<sexp>, <count_nexp>)

Return the left-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 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.

41.10 Lower$

Syntax: Lower$(<sexp>)

Returns <sexp> in all lower case characters.

41.11 Ltrim$

Syntax: Ltrim$(<sexp>{, <test_sexp>})

Exactly like **Trim$()**, except that **Ltrim$()** trims only the left end of the source string <sexp>.

41.12 Mid$

Syntax: Mid$(<sexp>, <start_nexp>{, <count_nexp>})

Return a substring of the string <sexp>, 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 ""
```

41.13 Oct$

Syntax: Oct$(<nexp>)

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

41.14 Replace$

Syntax: Replace$(<sexp>, <argument_sexp>, <replace_sexp>)

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

41.15 Right$

Syntax: 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.

41.16 Rtrim$

Syntax: Rtrim$(<sexp>{, <test_sexp>})

Exactly like **Trim$()**, except that **Rtrim$()** trims only the right end of the source string <sexp>.

41.17 Str$

Syntax: Str$(<nexp>)

Returns the string representation of <nexp>.

41.18 Trim$

Syntax: 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.

41.19 Upper$

Syntax: Upper$(<sexp>)

Returns <nexp> in all upper case characters.

41.20 Using$

Syntax: 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 at http://developer.android.com/reference/java/util/Formatter.html, also at 52 Appendix – Formatter.

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.

41.20.1 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.

41.20.2 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.

41.20.2.1 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                                                % sample 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.

41.20.2.2 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>	

41.20.2.3 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.

41.20.2.4 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)

41.20.2.5 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.

41.20.2.6 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" prints " foo"
"%-4s",  "foo" prints "foo "
"%4.2s", "foo" prints "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"
```

41.20.3 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()
```

41.21 Word$

Syntax: 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.

42 String Functions That Return a Number

42.1 Ascii

Syntax: 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.

42.2 Bin

Syntax: 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.

42.3 Ends_with

Syntax: 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.

42.4 Hex

Syntax: 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.

42.5 Is_in

Syntax: 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.

42.6 Is_number

Syntax: 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.

42.7 Len

Syntax: Len(<sexp>)

Returns the length of the <sexp>.

42.8 Oct

Syntax: 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.

42.9 Starts_with

Syntax: 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.

42.10 Ucode

Syntax: 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()**.

42.11 Val

Syntax: 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.

43 String Commands

See also the various String Functions.

43.1 Decrypt

Syntax: 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 encryption algorithm used is "PBEWithMD5AndDES".

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>)**.

43.2 Encrypt

Syntax: 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 encryption algorithm used is "PBEWithMD5AndDES".

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>)**.

43.3 Join / Join.all

Syntax: Join <source_array$[]>, <result_svar> {, <separator_sexp>{, <wrapper_sexp}}

or

Syntax: Join.all <source_array$[]>, <result_svar> {, <separator_sexp>{, <wrapper_sexp}}

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.

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.

43.4 Split / Split.all

Syntax: Split <result_array$[]>, <sexp> {, <test_sexp>}

or

Syntax: Split.all <result_array$[]>, <sexp> {, <test_sexp>}

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.

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

44 Superuser Commands

See comments under chapter 45, System Commands. The Su commands are identical to the System commands except that for Su commands:

- The initial working directory is the root directory, /.
- The resulting command shell has superuser privileges.

Some devices do not allow the SU.OPEN statement to execute successfully. In this case, the statement fails, issuing an error message such as the following:

```
SU Exception: Cannot run program "su": error=13, Permission denied
```

44.1 Su.close

Syntax: Su.close

Exits the Superuser mode.

44.2 Su.open

Syntax: 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.

44.3 Su.read.line

Syntax: Su.read.line <svar>

Places the next available response line into the string variable.

44.4 Su.read.ready

Syntax: 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.

44.5 Su.write

Syntax: Su.write <sexp>

Executes a Superuser command.

45 System Commands

BASIC! runs on devices that use Android, a Linux-based operating system. System commands allow BASIC! to send text commands directly to the operating system. These commands are essentially Linux commands.

The BASIC! statements in this section open a command shell. This is an environment with a saved context. The context remembers the results of a given command so that they affect subsequent commands. For example, if you change the working directory of a shell, the change remains in effect for subsequent statements.

The BASIC! program must open the command interface before using it, and should close it when operations are complete. The command interface works by sending string expressions and then waiting for responses to be read back, one line at a time, into string variables. The time this takes is device dependent. The shell does not respond to some commands at all, so the BASIC! program should loop and be ready to time-out in an orderly manner if the shell does not respond.

The System statements use a shell to the Android operating system. The SU statements use a shell with superuser privileges. Only one can be active at any time.

45.1 System.close

Syntax: System.close

Exits the System Shell mode. The shell's environment and context is discarded. Opening a new shell, with the System or SU statements, will not have any information from a previous shell, unless the program has saved it in, for example, a file or database.

45.2 System.open

Syntax: 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.

45.3 System.read.line

Syntax: System.read.line <svar>

Places the next available response line into the string variable. The returned string does not include a line terminator.

The BASIC! program should use System.read.ready statement to see if a response is available. If a response is not available, System.read.line sets <svar> to an empty string.

If System.read.ready indicates that the command shell has responded, then System.read.line can be called repeatedly, without further polling or pauses, to retrieve each line of the response in sequence. The BASIC! program should be written to take into account how many lines a command will return, or continue calling System.read.line until it returns an empty string or a string known to be the last line of the response. Some commands return blank lines; these also cause System.read.line to set <svar> to an empty string.

45.4 System.read.ready <nvar>

Syntax: 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 period of time (maybe a second) then it should be assumed that there will be no response.

45.5 System.write

Syntax: System.write <sexp>

Sends a System command <sexp> to the shell. The command in the string does not need to end with a line terminator.

This example will request the listing of a specified directory:

```
SYSTEM.WRITE "ls -al " + DirectoryName$
```

46 Time Functions

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.

46.1 Clock

Syntax: Clock()

Returns the time in milliseconds since the last boot.

46.2 Time

Syntax: Time()

or

Syntax: Time(<year_exp>, <month_exp>, <day_exp>, <hour_exp>, <minute_exp>, <second_exp>)

The first form of this command 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.

The second form of this command uses parameters to 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.

47 Time Commands

47.1 Time

Syntax: 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
```

47.2 TimeZone.get

Syntax: 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.

47.3 TimeZone.list

Syntax: 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.

47.4 TimeZone.set

Syntax: 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).

48 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.

48.1 OnTimer:

Interrupt handler for the timer interrupt. When done, execute the **Timer.resume** command to resume the interrupted program.

48.2 Timer.clear

Syntax: Timer.clear

Clears the repeating timer. No further timer interrupts will occur.

48.3 Timer.resume

Syntax: Timer.resume

This command resumes an interrupted program. It should be included in an interrupt handler as described in section **OnTimer:**.

48.4 Timer.set

Syntax: 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.

Example:

```
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
```

49 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-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.

49.1 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).

49.2 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.

49.3 Commands

49.3.1 Fn.def

Syntax: 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 π = 3!
```

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

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 called 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.

49.3.2 Fn.end

Syntax: 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 ("").

49.3.3 Fn.rtn

Syntax: 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 function.

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.

49.3.4 Call

Syntax: 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** for details.

50 Appendix - Supported media formats

This document describes the media codec, container, and network protocol support provided by the Android platform.

The tables below describe the media format support built into the Android platform. YES means the format is available on handhelds and tablets running all Android versions. Where a specific Android platform is specified, the format is available on handsets and tablets running that version and all later versions. The format might also be available in earlier versions, but this is not guaranteed. On form factors other than handsets and tablets, media format support may vary.

Note that a particular mobile device might support additional formats or file types that are not listed in these tables. In addition, if you use a Media Codec directly, you can access any of the available media formats regardless of the supported file types and container formats.

50.1 Audio support

Format	Encoder	Decoder	Details	File Types Container Formats
AAC LC	YES	YES	Support for mono/stereo/5.0/5.1 content with standard sampling rates from 8 to 48 kHz.	• 3GPP (.3gp) • MPEG-4 (.mp4, .m4a) • ADTS raw AAC (.aac, decode in Android 3.1+, encode in Android 4.0+, ADIF not supported) • MPEG-TS (.ts, not seekable, Android 3.0+)
HE-AACv1 (AAC+)	Android 4.1+	YES		
HE-AACv2 (enhanced AAC+)		YES	Support for stereo/5.0/5.1 content with standard sampling rates from 8 to 48 kHz.	
xHE-AAC		Android 9+	Support for up to 8ch content with standard sampling rates from 8 to 48 kHz	
AAC ELD (enhanced low delay AAC)	Android 4.1+	Android 4.1+	Support for mono/stereo content with standard sampling rates from 16 to 48 kHz	
AMR-NB	YES	YES	4.75 to 12.2 kbps sampled @ 8kHz	• 3GPP (.3gp) • AMR (.amr)
AMR-WB	YES	YES	9 rates from 6.60 kbit/s to 23.85 kbit/s sampled @ 16kHz	
FLAC	Android 4.1+	Android 3.1+	Mono/Stereo (no multichannel). Sample rates up to 48 kHz (but up to 44.1 kHz is recommended on devices with 44.1 kHz output, as the 48 to 44.1 kHz downsampler does not include a low-pass filter). 16-bit recommended; no dither applied for 24-bit.	• FLAC (.flac) • MPEG-4 (.mp4, .m4a, Android 10+)
MIDI		YES	MIDI Type 0 and 1. DLS Version 1 and 2. XMF and	• Type 0 and 1 (.mid, .xmf, .mxmf)

			Mobile XMF. Support for ringtone formats RTTTL/RTX, OTA, and iMelody	• RTTTL/RTX (.rtttl, .rtx) • OTA (.ota) • iMelody (.imy)
MP3		YES	Mono/Stereo 8-320Kbps constant (CBR) or variable bit-rate (VBR)	• MP3 (.mp3) • MPEG-4 (.mp4, .m4a, Android 10+) • Matroska (.mkv, Android 10+)
Opus	Android 10+	Android 5.0+		• Ogg (.ogg) • Matroska (.mkv)
PCM/WAVE	Android 4.1+	YES	8- and 16-bit linear PCM (rates up to limit of hardware). Sampling rates for raw PCM recordings at 8000, 16000 and 44100 Hz.	WAVE (.wav)
Vorbis		YES		• Ogg (.ogg) • Matroska (.mkv, Android 4.0+) • MPEG-4 (.mp4, .m4a, Android 10+)

51 Appendix - urlencoded

Form content type: application/x-www-form-urlencoded

This is the default content type. Forms submitted with this content type must be encoded as follows:

1. Control names and values are escaped. Space characters are replaced by `+', and then reserved characters are escaped as described in [RFC1738], section 2.2. Non-printing characters are replaced by "%HH', a percent sign and two hexadecimals representing the ASCII code of the character. Line breaks are represented as "CR LF" pairs (i.e., "%0D%0A').
2. The control names/values are listed in the order they appear in the document. The name is separated from the value by "=' and name/value pairs are separated from each other by "&'.

52 Appendix – Formatter

52.1 Formatter

An interpreter for `printf` style format strings. This class provides support for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output. Common Java types such as `byte`, `BigDecimal`, and `Calendar` are supported. Limited formatting customization for arbitrary user types is provided through the `Formattable` interface.

Formatters are not necessarily safe for multi-threaded access. Thread safety is optional and is the responsibility of users of methods in this class.

Formatted printing for the Java language is heavily inspired by C's `printf`. Although the format strings are similar to C, some customizations have been made to accommodate the Java language and exploit some of its features. Also, Java formatting is more strict than C's; for example, if a conversion is incompatible with a flag, an exception will be thrown. In C inapplicable flags are silently ignored. The format strings are thus intended to be recognizable to C programmers but not necessarily completely compatible with those in C.

Examples of expected usage:

```
StringBuilder sb = new StringBuilder();
// Send all output to the Appendable object sb
Formatter formatter = new Formatter(sb, Locale.US);

// Explicit argument indices may be used to re-order output.
formatter.format("%4$2s %3$2s %2$2s %1$2s", "a", "b", "c", "d")
// -> " d  c  b  a"

// Optional locale as the first argument can be used to get
// locale-specific formatting of numbers.  The precision and width can be
// given to round and align the value.
formatter.format(Locale.FRANCE, "e = %+10.4f", Math.E);
// -> "e =    +2,7183"

// The '(' numeric flag may be used to format negative numbers with
// parentheses rather than a minus sign.  Group separators are
// automatically inserted.
formatter.format("Amount gained or lost since last statement: $ %(,.2f",
balanceDelta);
// -> "Amount gained or lost since last statement: $ (6,217.58)"
```

Convenience methods for common formatting requests exist as illustrated by the following invocations:

```
// Writes a formatted string to System.out.
System.out.format("Local time: %tT", Calendar.getInstance());
// -> "Local time: 13:34:18"

// Writes formatted output to System.err.
System.err.printf("Unable to open file '%1$s': %2$s",
fileName, exception.getMessage());
// -> "Unable to open file 'food': No such file or directory"
```

Like C's `sprintf(3)`, Strings may be formatted using the static method `String.format`:

```
// Format a string containing a date.
import java.util.Calendar;
import java.util.GregorianCalendar;
import static java.util.Calendar.*;

Calendar c = new GregorianCalendar(1995, MAY, 23);
String s = String.format("Duke's Birthday: %1$tb %1$te, %1$tY", c);
// -> s == "Duke's Birthday: May 23, 1995"
```

52.2 Format String Syntax

Every method which produces formatted output requires a *format string* and an *argument list*. The format string is a `String` which may contain fixed text and one or more embedded *format specifiers*. Consider the following example:

```
Calendar c = ...;
String s = String.format("Duke's Birthday: %1$tm %1$te,%1$tY", c);
```

This format string is the first argument to the `format` method. It contains three format specifiers "`%1$tm`", "`%1$te`", and "`%1$tY`" which indicate how the arguments should be processed and where they should be inserted in the text. The remaining portions of the format string are fixed text including `"Dukes Birthday: "` and any other spaces or punctuation. The argument list consists of all arguments passed to the method after the format string. In the above example, the argument list is of size one and consists of the `Calendar` object `c`.

The format specifiers for general, character, and numeric types have the following syntax:

```
%[argument_index$][flags][width][.precision]conversion
```

The optional *argument_index* is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by "`1$`", the second by "`2$`", etc.

The optional *flags* is a set of characters that modify the output format. The set of valid flags depends on the conversion.

The optional *width* is a positive decimal integer indicating the minimum number of characters to be written to the output.

The optional *precision* is a non-negative decimal integer usually used to restrict the number of characters. The specific behavior depends on the conversion.

The required *conversion* is a character indicating how the argument should be formatted. The set of valid conversions for a given argument depends on the argument's data type.

The format specifiers for types which are used to represents dates and times have the following syntax:

```
%[argument_index$][flags][width]conversion
```

The optional *argument_index*, *flags* and *width* are defined as above.

The required *conversion* is a two character sequence. The first character is `'t'` or `'T'`. The second character indicates the format to be used. These characters are similar to but not completely identical to those defined by GNU `date` and POSIX `strftime(3c)`.

The format specifiers which do not correspond to arguments have the following syntax:

```
%[flags][width]conversion
```

The optional *flags* and *width* is defined as above.

The required *conversion* is a character indicating content to be inserted in the output.

52.3 Conversions

Conversions are divided into the following categories:

1. **General** - may be applied to any argument type.
2. **Character** - may be applied to basic types which represent Unicode characters: `char`, `Character`, `byte`, `Byte`, `short`, and `Short`. This conversion may also be applied to the types `int` and `Integer` when `Character.isValidCodePoint(int)` returns `true`.
3. **Numeric**
    1. **Integral** - may be applied to Java integral types: `byte`, `Byte`, `short`, `Short`, `int` and `Integer`, `long`, `Long`, and `BigInteger` (but not `char` or `Character`).
    2. **Floating Point** - may be applied to Java floating-point types: `float`, `Float`, `double`, `Double`, and `BigDecimal`.
4. **Date/Time** - may be applied to Java types which are capable of encoding a date or time: `long`, `Long`, `Calendar`, `Date` and `TemporalAccessor`.
5. **Percent** - produces a literal `'%'` (`'\u0025'`).
6. **Line Separator** - produces the platform-specific line separator.

The following table summarizes the supported conversions. Conversions denoted by an upper-case character (i.e. `'B'`, `'H'`, `'S'`, `'C'`, `'X'`, `'E'`, `'G'`, `'A'`, and `'T'`) are the same as those for the corresponding lower-case conversion characters except that the result is converted to upper case according to the rules of the prevailing `Locale`. The result is equivalent to the following invocation of `String.toUpperCase()`.

out.toUpperCase()

Conversion	Argument Category	Description
`'b'`, `'B'`	general	If the argument *arg* is `null`, then the result is "`false`". If *arg* is a `boolean` or `Boolean`, then the result is the string returned by `String.valueOf(arg)`. Otherwise, the result is "true".

`'h'`, `'H'`	general	If the argument *arg* is `null`, then the result is "`null`". Otherwise, the result is obtained by invoking `Integer.toHexString(arg.hashCode())`.
`'s'`, `'S'`	general	If the argument *arg* is `null`, then the result is "`null`". If *arg* implements `Formattable`, then `arg.formatTo` is invoked. Otherwise, the result is obtained by invoking `arg.toString()`.
`'c'`, `'C'`	character	The result is a Unicode character.
`'d'`	integral	The result is formatted as a decimal integer.
`'o'`	integral	The result is formatted as an octal integer.
`'x'`, `'X'`	integral	The result is formatted as a hexadecimal integer.
`'e'`, `'E'`	floating point	The result is formatted as a decimal number in computerized scientific notation.
`'f'`	floating point	The result is formatted as a decimal number.
`'g'`, `'G'`	floating point	The result is formatted using computerized scientific notation or decimal format, depending on the precision and the value after rounding.
`'a'`, `'A'`	floating point	The result is formatted as a hexadecimal floating-point number with a significand and an exponent. This conversion is **not** supported for the `BigDecimal` type despite the latter's being in the *floating point* argument category.
`'t'`, `'T'`	date/time	Prefix for date and time conversion characters. See Date/Time Conversions.
`'%'`	percent	The result is a literal `'%'` (`'\u0025'`).
`'n'`	line separator	The result is the platform-specific line separator.

Any characters not explicitly defined as conversions are illegal and are reserved for future extensions.

52.3.1 Date/Time Conversions

The following date and time conversion suffix characters are defined for the `'t'` and `'T'` conversions. The types are similar to but not completely identical to those defined by GNU `date` and POSIX `strftime(3c)`. Additional conversion types are provided to access Java-specific functionality (e.g. `'L'` for milliseconds within the second).

The following conversion characters are used for formatting times:

`'H'`	Hour of the day for the 24-hour clock, formatted as two digits with a leading zero as necessary i.e. `00 - 23`.
`'I'`	Hour for the 12-hour clock, formatted as two digits with a leading zero as necessary, i.e. `01 -`

	12.
`'k'`	Hour of the day for the 24-hour clock, i.e. `0 - 23`.
`'l'`	Hour for the 12-hour clock, i.e. `1 - 12`.
`'M'`	Minute within the hour formatted as two digits with a leading zero as necessary, i.e. `00 - 59`.
`'S'`	Seconds within the minute, formatted as two digits with a leading zero as necessary, i.e. `00 - 60` ("`60`" is a special value required to support leap seconds).
`'L'`	Millisecond within the second formatted as three digits with leading zeros as necessary, i.e. `000 - 999`.
`'N'`	Nanosecond within the second, formatted as nine digits with leading zeros as necessary, i.e. `000000000 - 999999999`.
`'p'`	Locale-specific morning or afternoon marker in lower case, e.g."`am`" or "`pm`". Use of the conversion prefix `'T'` forces this output to upper case.
`'z'`	RFC 822 style numeric time zone offset from GMT, e.g. `-0800`. This value will be adjusted as necessary for Daylight Saving Time. For `long`, `Long`, and `Date` the time zone used is the default time zone for this instance of the Java virtual machine.
`'Z'`	A string representing the abbreviation for the time zone. This value will be adjusted as necessary for Daylight Saving Time. For `long`, `Long`, and `Date` the time zone used is the default time zone for this instance of the Java virtual machine. The Formatter's locale will supersede the locale of the argument (if any).
`'s'`	Seconds since the beginning of the epoch starting at 1 January 1970 `00:00:00` UTC, i.e. `Long.MIN_VALUE/1000` to `Long.MAX_VALUE/1000`.
`'Q'`	Milliseconds since the beginning of the epoch starting at 1 January 1970 `00:00:00` UTC, i.e. `Long.MIN_VALUE` to `Long.MAX_VALUE`.

The following conversion characters are used for formatting dates:

`'B'`	Locale-specific full month name, e.g. `"January"`, `"February"`.
`'b'`	Locale-specific abbreviated month name, e.g. `"Jan"`, `"Feb"`.
`'h'`	Same as `'b'`.
`'A'`	Locale-specific full name of the day of the week, e.g. `"Sunday"`, `"Monday"`.
`'a'`	Locale-specific short name of the day of the week, e.g. `"Sun"`, `"Mon"`.
`'C'`	Four-digit year divided by `100`, formatted as two digits with leading zero as necessary, i.e. `00 - 99`.
`'Y'`	Year, formatted as at least four digits with leading zeros as necessary, e.g. `0092` equals `92` CE for the Gregorian calendar.
`'y'`	Last two digits of the year, formatted with leading zeros as necessary, i.e. `00 - 99`.

`'j'`	Day of year, formatted as three digits with leading zeros as necessary, e.g. `001 - 366` for the Gregorian calendar.
`'m'`	Month, formatted as two digits with leading zeros as necessary, i.e. `01 - 13`.
`'d'`	Day of month, formatted as two digits with leading zeros as necessary, i.e. `01 - 31`.
`'e'`	Day of month, formatted as two digits, i.e. `1 - 31`.

The following conversion characters are used for formatting common date/time compositions.

`'R'`	Time formatted for the 24-hour clock as `"%tH:%tM"`.
`'T'`	Time formatted for the 24-hour clock as `"%tH:%tM:%tS"`.
`'r'`	Time formatted for the 12-hour clock as `"%tI:%tM:%tS %Tp"`. The location of the morning or afternoon marker (`'%Tp'`) may be locale-dependent.
`'D'`	Date formatted as `"%tm/%td/%ty"`.
`'F'`	ISO 8601 complete date formatted as `"%tY-%tm-%td"`.
`'c'`	Date and time formatted as `"%ta %tb %td %tT %tZ %tY"`, e.g. `"Sun Jul 20 16:17:00 EDT 1969"`.

Any characters not explicitly defined as date/time conversion suffixes are illegal and are reserved for future extensions.

52.3.2 Flags

The following table summarizes the supported flags. y means the flag is supported for the indicated argument types.

Flag	General	Character	Integral	Floating Point	Date/ Time	Description
'-'	y	y	y	y	y	The result will be left-justified.
'#'	y[1]	-	y[3]	y	-	The result should use a conversion-dependent alternate form.
'+'	-	-	y[4]	y	-	The result will always include a sign.
' '	-	-	y[4]	y	-	The result will include a leading space for positive values.
'0'	-	-	y	y	-	The result will be zero-padded.
','	-	-	y[2]	y[5]	-	The result will include locale-specific grouping separators.
'('	-	-	y[4]	y[5]	-	The result will enclose negative numbers in parentheses.

[1] Depends on the definition of `Formattable`.

[2] For `'d'` conversion only.

[3] For `'o'`, `'x'`, and `'X'` conversions only.

[4] For `'d'`, `'o'`, `'x'`, and `'X'` conversions applied to `BigInteger` or `'d'` applied to `byte`, `Byte`, `short`, `Short`, `int` and `Integer`, `long`, and `Long`.

[5] For `'e'`, `'E'`, `'f'`, `'g'`, and `'G'` conversions only.

Any characters not explicitly defined as flags are illegal and are reserved for future extensions.

52.3.3 Width

The width is the minimum number of characters to be written to the output. For the line separator conversion, width is not applicable; if it is provided, an exception will be thrown.

52.3.4 Precision

For general argument types, the precision is the maximum number of characters to be written to the output.

For the floating-point conversions `'a'`, `'A'`, `'e'`, `'E'`, and `'f'` the precision is the number of digits after the radix point. If the conversion is `'g'` or `'G'`, then the precision is the total number of digits in the resulting magnitude after rounding.

For character, integral, and date/time argument types and the percent and line separator conversions, the precision is not applicable; if a precision is provided, an exception will be thrown.

52.3.5 Argument Index

The argument index is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by "`1$`", the second by "`2$`", etc.

Another way to reference arguments by position is to use the `'<'` (`'\u003c'`) flag, which causes the argument for the previous format specifier to be re-used. For example, the following two statements would produce identical strings:

```
   Calendar c = ...;
   String s1 = String.format("Duke's Birthday: %1$tm %1$te,%1$tY", c);

   String s2 = String.format("Duke's Birthday: %1$tm %<te,%<tY", c);
```

52.4 Details

This section is intended to provide behavioral details for formatting, including conditions and exceptions, supported data types, localization, and interactions between flags, conversions, and data types.

Any characters not explicitly defined as conversions, date/time conversion suffixes, or flags are illegal and are reserved for future extensions. Use of such a character in a format string will cause an `UnknownFormatConversionException` or `UnknownFormatFlagsException` to be thrown.

If the format specifier contains a width or precision with an invalid value or which is otherwise unsupported, then a `IllegalFormatWidthException` or `IllegalFormatPrecisionException` respectively will be thrown.

If a format specifier contains a conversion character that is not applicable to the corresponding argument, then an `IllegalFormatConversionException` will be thrown.

All specified exceptions may be thrown by any of the `format` methods of `Formatter` as well as by any `format` convenience methods such as `String.format` and `PrintStream.printf`.

Conversions denoted by an upper-case character (i.e. `'B'`, `'H'`, `'S'`, `'C'`, `'X'`, `'E'`, `'G'`, `'A'`, and `'T'`) are the same as those for the corresponding lower-case conversion characters except that the result is converted to upper case according to the rules of the prevailing `Locale`. The result is equivalent to the following invocation of `String#toUpperCase()`.

out.toUpperCase()

52.4.1 General

The following general conversions may be applied to any argument type:

`'b'`	`'\u0062'`	Produces either "`true`" or "`false`" as returned by `Boolean#toString(boolean)`. If the argument is `null`, then the result is "`false`". If the argument is a `boolean` or `Boolean`, then the result is the string returned by `String.valueOf()`. Otherwise, the result is "`true`". If the `'#'` flag is given, then a `FormatFlagsConversionMismatchException` will be thrown.
`'B'`	`'\u0042'`	The upper-case variant of `'b'`.
`'h'`	`'\u0068'`	Produces a string representing the hash code value of the object. If the argument, *arg* is `null`, then the result is "`null`". Otherwise, the result is obtained by invoking `Integer.toHexString(arg.hashCode())`. If the `'#'` flag is given, then a `FormatFlagsConversionMismatchException` will be thrown.
`'H'`	`'\u0048'`	The upper-case variant of `'h'`.
`'s'`	`'\u0073'`	Produces a string. If the argument is `null`, then the result is "`null`". If the argument implements `Formattable`, then its `formatTo` method is invoked. Otherwise, the result is obtained by invoking the argument's `toString()` method. If the `'#'` flag is given and the argument is not a `Formattable` , then a `FormatFlagsConversionMismatchException` will be thrown.
`'S'`	`'\u0053'`	The upper-case variant of `'s'`.

The following flags apply to general conversions:

`'-'`	`'\u002d'`	Left justifies the output. Spaces (`'\u0020'`) will be added at the end of the converted value as required to fill the minimum width of the field. If the width is

		not provided, then a `MissingFormatWidthException` will be thrown. If this flag is not given then the output will be right-justified.
`'#'`	`'\u0023'`	Requires the output use an alternate form. The definition of the form is specified by the conversion.

The width is the minimum number of characters to be written to the output. If the length of the converted value is less than the width then the output will be padded by `' '` (`'\u0020'`) until the total number of characters equals the width. The padding is on the left by default. If the `'-'` flag is given, then the padding will be on the right. If the width is not specified then there is no minimum.

The precision is the maximum number of characters to be written to the output. The precision is applied before the width, thus the output will be truncated to `precision` characters even if the width is greater than the precision. If the precision is not specified then there is no explicit limit on the number of characters.

52.4.2 Character

This conversion may be applied to `char` and `Character`. It may also be applied to the types `byte`, `Byte`, `short`, and `Short`, `int` and `Integer` when `Character#isValidCodePoint` returns `true`. If it returns `false` then an `IllegalFormatCodePointException` will be thrown.

`'c'`	`'\u0063'`	Formats the argument as a Unicode character as described in Unicode Character Representation. This may be more than one 16-bit `char` in the case where the argument represents a supplementary character. If the `'#'` flag is given, then a `FormatFlagsConversionMismatchException` will be thrown.
`'C'`	`'\u0043'`	The upper-case variant of `'c'`.

The `'-'` flag defined for General conversions applies. If the `'#'` flag is given, then a `FormatFlagsConversionMismatchException` will be thrown.

The width is defined as for General conversions.

The precision is not applicable. If the precision is specified then an `IllegalFormatPrecisionException` will be thrown.

52.4.3 Numeric

Numeric conversions are divided into the following categories:

1. Byte, Short, Integer, and Long
2. BigInteger
3. Float and Double
4. BigDecimal

Numeric types will be formatted according to the following algorithm:

Number Localization Algorithm

After digits are obtained for the integer part, fractional part, and exponent (as appropriate for the data type), the following transformation is applied:

1. Each digit character *d* in the string is replaced by a locale-specific digit computed relative to the current locale's zero digit *z*; that is *d* - `'0'` + *z*.
2. If a decimal separator is present, a locale-specific decimal separator is substituted.
3. If the `','` (`'\u002c'`) flag is given, then the locale-specific grouping separator is inserted by scanning the integer part of the string from least significant to most significant digits and inserting a separator at intervals defined by the locale's grouping size.
4. If the `'0'` flag is given, then the locale-specific zero digits are inserted after the sign character, if any, and before the first non-zero digit, until the length of the string is equal to the requested field width.
5. If the value is negative and the `'('` flag is given, then a `'('` (`'\u0028'`) is prepended and a `')'` (`'\u0029'`) is appended.
6. If the value is negative (or floating-point negative zero) and `'('` flag is not given, then a `'-'` (`'\u002d'`) is prepended.
7. If the `'+'` flag is given and the value is positive or zero (or floating-point positive zero), then a `'+'` (`'\u002b'`) will be prepended.

If the value is NaN or positive infinity the literal strings "NaN" or "Infinity" respectively, will be output. If the value is negative infinity, then the output will be "(Infinity)" if the `'('` flag is given otherwise the output will be "-Infinity". These values are not localized.

Byte, Short, Integer, and Long

The following conversions may be applied to `byte`, `Byte`, `short`, `Short`, `int` and `Integer`, `long`, and `Long`.

`'d'`	`'\u0064'`	Formats the argument as a decimal integer. The localization algorithm is applied. If the `'0'` flag is given and the value is negative, then the zero padding will occur after the sign. If the `'#'` flag is given then a `FormatFlagsConversionMismatchException` will be thrown.
`'o'`	`'\u006f'`	Formats the argument as an integer in base eight. No localization is applied. If *x* is negative then the result will be an unsigned value generated by adding 2^n to the value where `n` is the number of bits in the type as returned by the static `SIZE` field in the Byte, Short, Integer, or Long classes as appropriate. If the `'#'` flag is given then the output will always begin with the radix indicator `'0'`. If the `'0'` flag is given then the output will be padded with leading zeros to the field width following any indication of sign. If `'('`, `'+'`, `' '`, or `','` flags are given then a `FormatFlagsConversionMismatchException` will be thrown.
`'x'`	`'\u0078'`	Formats the argument as an integer in base sixteen. No localization is applied.

		If *x* is negative then the result will be an unsigned value generated by adding 2^n to the value where `n` is the number of bits in the type as returned by the static `SIZE` field in the Byte, Short, Integer, or Long classes as appropriate. If the `'#'` flag is given then the output will always begin with the radix indicator `"0x"`. If the `'0'` flag is given then the output will be padded to the field width with leading zeros after the radix indicator or sign (if present). If `'('`, `' '`, `'+'`, or `','` flags are given then a `FormatFlagsConversionMismatchException` will be thrown.
`'X'`	`'\u0058'`	The upper-case variant of `'x'`. The entire string representing the number will be converted to upper case including the `'x'` (if any) and all hexadecimal digits `'a'` - `'f'` (`'\u0061'` - `'\u0066'`).

If the conversion is `'o'`, `'x'`, or `'X'` and both the `'#'` and the `'0'` flags are given, then result will contain the radix indicator (`'0'` for octal and `"0x"` or `"0X"` for hexadecimal), some number of zeros (based on the width), and the value.

If the `'-'` flag is not given, then the space padding will occur before the sign.

The following flags apply to numeric integral conversions:

`'+'`	`'\u002b'`	Requires the output to include a positive sign for all positive numbers. If this flag is not given then only negative values will include a sign. If both the `'+'` and `' '` flags are given then an `IllegalFormatFlagsException` will be thrown.
`' '`	`'\u0020'`	Requires the output to include a single extra space (`'\u0020'`) for non-negative values. If both the `'+'` and `' '` flags are given then an `IllegalFormatFlagsException` will be thrown.
`'0'`	`'\u0030'`	Requires the output to be padded with leading zeros to the minimum field width following any sign or radix indicator except when converting NaN or infinity. If the width is not provided, then a `MissingFormatWidthException` will be thrown. If both the `'-'` and `'0'` flags are given then an `IllegalFormatFlagsException` will be thrown.
`','`	`'\u002c'`	Requires the output to include the locale-specific group separators as described in the "group" section of the localization algorithm.
`'('`	`'\u0028'`	Requires the output to prepend a `'('` (`'\u0028'`) and append a `')'` (`'\u0029'`) to negative values.

If no flags are given the default formatting is as follows:

- The output is right-justified within the `width`.
- Negative numbers begin with a `'-'` (`'\u002d'`).
- Positive numbers and zero do not include a sign or extra leading space.
- No grouping separators are included.

The width is the minimum number of characters to be written to the output. This includes any signs, digits, grouping separators, radix indicator, and parentheses. If the length of the converted value is less than the width then the output will be padded by spaces (`'\u0020'`) until the total number of characters equals width. The padding is on the left by default. If `'-'` flag is given then the padding will be on the right. If width is not specified then there is no minimum.

The precision is not applicable. If precision is specified then an `IllegalFormatPrecisionException` will be thrown.

BigInteger

The following conversions may be applied to `BigInteger`.

`'d'`	`'\u0064'`	Requires the output to be formatted as a decimal integer. The localization algorithm is applied. If the `'#'` flag is given `FormatFlagsConversionMismatchException` will be thrown.
`'o'`	`'\u006f'`	Requires the output to be formatted as an integer in base eight. No localization is applied. If *x* is negative then the result will be a signed value beginning with `'-'` (`'\u002d'`). Signed output is allowed for this type because unlike the primitive types it is not possible to create an unsigned equivalent without assuming an explicit data-type size. If *x* is positive or zero and the `'+'` flag is given then the result will begin with `'+'` (`'\u002b'`). If the `'#'` flag is given then the output will always begin with `'0'` prefix. If the `'0'` flag is given then the output will be padded with leading zeros to the field width following any indication of sign. If the `','` flag is given then a `FormatFlagsConversionMismatchException` will be thrown.
`'x'`	`'\u0078'`	Requires the output to be formatted as an integer in base sixteen. No localization is applied. If *x* is negative then the result will be a signed value beginning with `'-'` (`'\u002d'`). Signed output is allowed for this type because unlike the primitive types it is not possible to create an unsigned equivalent without assuming an explicit data-type size. If *x* is positive or zero and the `'+'` flag is given then the result will begin with `'+'`

		(`'\u002b'`). If the `'#'` flag is given then the output will always begin with the radix indicator `"0x"`. If the `'0'` flag is given then the output will be padded to the field width with leading zeros after the radix indicator or sign (if present). If the `','` flag is given then a `FormatFlagsConversionMismatchException` will be thrown.
`'X'`	`'\u0058'`	The upper-case variant of `'x'`. The entire string representing the number will be converted to upper case including the `'x'` (if any) and all hexadecimal digits `'a'` - `'f'` (`'\u0061'` - `'\u0066'`).

If the conversion is `'o'`, `'x'`, or `'X'` and both the `'#'` and the `'0'` flags are given, then result will contain the base indicator (`'0'` for octal and `"0x"` or `"0X"` for hexadecimal), some number of zeros (based on the width), and the value.

If the `'0'` flag is given and the value is negative, then the zero padding will occur after the sign.

If the `'-'` flag is not given, then the space padding will occur before the sign.

All flags defined for Byte, Short, Integer, and Long apply. The default behavior when no flags are given is the same as for Byte, Short, Integer, and Long.

The specification of width is the same as defined for Byte, Short, Integer, and Long.

The precision is not applicable. If precision is specified then an `IllegalFormatPrecisionException` will be thrown.

Float and Double

The following conversions may be applied to `float`, `Float`, `double` and `Double`.

`'e'`	`'\u0065'`	Requires the output to be formatted using computerized scientific notation. The localization algorithm is applied. The formatting of the magnitude *m* depends upon its value. If *m* is NaN or infinite, the literal strings "NaN" or "Infinity", respectively, will be output. These values are not localized. If *m* is positive-zero or negative-zero, then the exponent will be `"+00"`. Otherwise, the result is a string that represents the sign and magnitude (absolute value) of the argument. The formatting of the sign is described in the localization algorithm. The formatting of the magnitude *m* depends upon its value. Let *n* be the unique integer such that $10^n <= m < 10^{n+1}$; then let *a* be the mathematically exact quotient of *m* and 10^n so that $1 <= a < 10$. The magnitude is then represented as the integer part of *a*, as a single decimal digit, followed by the decimal separator followed by decimal digits representing the fractional part of *a*, followed by the lower-case locale-specific exponent separator (e.g. `'e'`), followed by the sign of the exponent, followed by a representation of *n* as a

		decimal integer, as produced by the method `Long#toString(long, int)`, and zero-padded to include at least two digits. The number of digits in the result for the fractional part of *m* or *a* is equal to the precision. If the precision is not specified then the default value is `6`. If the precision is less than the number of digits which would appear after the decimal point in the string returned by `Float#toString(float)` or `Double.toString(double)` respectively, then the value will be rounded using the round half up algorithm. Otherwise, zeros may be appended to reach the precision. For a canonical representation of the value, use `Float.toString(float)` or `Double#toString(double)` as appropriate. If the `','` flag is given, then an `FormatFlagsConversionMismatchException` will be thrown.
`'E'`	`'\u0045'`	The upper-case variant of `'e'`. The exponent symbol will be the upper-case locale-specific exponent separator (e.g. `'E'`).
`'g'`	`'\u0067'`	Requires the output to be formatted in general scientific notation as described below. The localization algorithm is applied. After rounding for the precision, the formatting of the resulting magnitude *m* depends on its value. If *m* is greater than or equal to 10^{-4} but less than $10^{\text{precision}}$ then it is represented in *decimal format*. If *m* is less than 10^{-4} or greater than or equal to $10^{\text{precision}}$, then it is represented in *computerized scientific notation*. The total number of significant digits in *m* is equal to the precision. If the precision is not specified, then the default value is `6`. If the precision is `0`, then it is taken to be `1`. If the `'#'` flag is given then an `FormatFlagsConversionMismatchException` will be thrown.
`'G'`	`'\u0047'`	The upper-case variant of `'g'`.
`'f'`	`'\u0066'`	Requires the output to be formatted using decimal format. The localization algorithm is applied. The result is a string that represents the sign and magnitude (absolute value) of the argument. The formatting of the sign is described in the localization algorithm. The formatting of the magnitude *m* depends upon its value. If *m* NaN or infinite, the literal strings "NaN" or "Infinity", respectively, will be output. These values are not localized. The magnitude is formatted as the integer part of *m*, with no leading zeroes, followed by the decimal separator followed by one or more decimal digits representing the fractional part of *m*. The number of digits in the result for the fractional part of *m* or *a* is equal to the precision. If the precision is not specified then the default value is `6`. If the precision is less than the number of digits which would appear after the decimal point in the string returned by `Float#toString(float)` or

		`Double.toString(double)` respectively, then the value will be rounded using the round half up algorithm. Otherwise, zeros may be appended to reach the precision. For a canonical representation of the value, use `Float.toString(float)` or `Double#toString(double)` as appropriate.
`'a'`	`'\u0061'`	Requires the output to be formatted in hexadecimal exponential form. No localization is applied. The result is a string that represents the sign and magnitude (absolute value) of the argument *x*. If *x* is negative or a negative-zero value then the result will begin with `'-'` (`'\u002d'`). If *x* is positive or a positive-zero value and the `'+'` flag is given then the result will begin with `'+'` (`'\u002b'`). The formatting of the magnitude *m* depends upon its value. • If the value is NaN or infinite, the literal strings "NaN" or "Infinity", respectively, will be output. • If *m* is zero then it is represented by the string `"0x0.0p0"`. • If *m* is a `double` value with a normalized representation then substrings are used to represent the significand and exponent fields. The significand is represented by the characters `"0x1."` followed by the hexadecimal representation of the rest of the significand as a fraction. The exponent is represented by `'p'` (`'\u0070'`) followed by a decimal string of the unbiased exponent as if produced by invoking `Integer.toString` on the exponent value. If the precision is specified, the value is rounded to the given number of hexadecimal digits. • If *m* is a `double` value with a subnormal representation then, unless the precision is specified to be in the range 1 through 12, inclusive, the significand is represented by the characters `'0x0.'` followed by the hexadecimal representation of the rest of the significand as a fraction, and the exponent represented by `'p-1022'`. If the precision is in the interval [1, 12], the subnormal value is normalized such that it begins with the characters `'0x1.'`, rounded to the number of hexadecimal digits of precision, and the exponent adjusted accordingly. Note that there must be at least one nonzero digit in a subnormal significand. If the `'('` or `','` flags are given, then a `FormatFlagsConversionMismatchException` will be thrown.
`'A'`	`'\u0041'`	The upper-case variant of `'a'`. The entire string representing the number will be converted to upper case including the `'x'` (`'\u0078'`) and `'p'` (`'\u0070'` and all hexadecimal digits `'a'` - `'f'` (`'\u0061'` - `'\u0066'`).

All flags defined for Byte, Short, Integer, and Long apply.

If the `'#'` flag is given, then the decimal separator will always be present.

If no flags are given the default formatting is as follows:

- The output is right-justified within the `width`.
- Negative numbers begin with a `'-'`.
- Positive numbers and positive zero do not include a sign or extra leading space.
- No grouping separators are included.
- The decimal separator will only appear if a digit follows it.

The width is the minimum number of characters to be written to the output. This includes any signs, digits, grouping separators, decimal separators, exponential symbol, radix indicator, parentheses, and strings representing infinity and NaN as applicable. If the length of the converted value is less than the width then the output will be padded by spaces (`'\u0020'`) until the total number of characters equals width. The padding is on the left by default. If the `'-'` flag is given then the padding will be on the right. If width is not specified then there is no minimum.

If the conversion is `'e'`, `'E'` or `'f'`, then the precision is the number of digits after the decimal separator. If the precision is not specified, then it is assumed to be `6`.

If the conversion is `'g'` or `'G'`, then the precision is the total number of significant digits in the resulting magnitude after rounding. If the precision is not specified, then the default value is `6`. If the precision is `0`, then it is taken to be `1`.

If the conversion is `'a'` or `'A'`, then the precision is the number of hexadecimal digits after the radix point. If the precision is not provided, then all of the digits as returned by `Double.toHexString(double)` will be output.

BigDecimal

The following conversions may be applied `BigDecimal`.

`'e'`	`'\u0065'`	Requires the output to be formatted using computerized scientific notation. The localization algorithm is applied. The formatting of the magnitude *m* depends upon its value. If *m* is positive-zero or negative-zero, then the exponent will be `"+00"`. Otherwise, the result is a string that represents the sign and magnitude (absolute value) of the argument. The formatting of the sign is described in the localization algorithm. The formatting of the magnitude *m* depends upon its value. Let *n* be the unique integer such that $10^n <= m < 10^{n+1}$; then let *a* be the mathematically exact quotient of *m* and 10^n so that $1 <= a < 10$. The magnitude is then represented as the integer part of *a*, as a single decimal digit, followed by the decimal separator followed by decimal digits representing the fractional part of *a*, followed by the exponent symbol `'e'` (`'\u0065'`), followed by the sign of the exponent, followed by a representation of *n* as a decimal integer, as produced by the method `Long#toString(long, int)`, and zero-padded to include at least two digits. The number of digits in the result for the fractional part of *m* or *a* is equal to the precision. If the precision is not specified then the default value is `6`. If the precision is less than the number of digits to the right of the decimal point then the value will be rounded using the round half up algorithm. Otherwise, zeros may be appended to reach the precision. For a canonical representation of the value,

		use `BigDecimal.toString()`. If the `','` flag is given, then an `FormatFlagsConversionMismatchException` will be thrown.
`'E'`	`'\u0045'`	The upper-case variant of `'e'`. The exponent symbol will be `'E'` (`'\u0045'`).
`'g'`	`'\u0067'`	Requires the output to be formatted in general scientific notation as described below. The localization algorithm is applied. After rounding for the precision, the formatting of the resulting magnitude *m* depends on its value. If *m* is greater than or equal to 10^{-4} but less than $10^{precision}$ then it is represented in *decimal format*. If *m* is less than 10^{-4} or greater than or equal to $10^{precision}$, then it is represented in *computerized scientific notation*. The total number of significant digits in *m* is equal to the precision. If the precision is not specified, then the default value is `6`. If the precision is `0`, then it is taken to be `1`. If the `'#'` flag is given then an `FormatFlagsConversionMismatchException` will be thrown.
`'G'`	`'\u0047'`	The upper-case variant of `'g'`.
`'f'`	`'\u0066'`	Requires the output to be formatted using decimal format. The localization algorithm is applied. The result is a string that represents the sign and magnitude (absolute value) of the argument. The formatting of the sign is described in the localization algorithm. The formatting of the magnitude *m* depends upon its value. The magnitude is formatted as the integer part of *m*, with no leading zeroes, followed by the decimal separator followed by one or more decimal digits representing the fractional part of *m*. The number of digits in the result for the fractional part of *m* or *a* is equal to the precision. If the precision is not specified then the default value is `6`. If the precision is less than the number of digits to the right of the decimal point then the value will be rounded using the round half up algorithm. Otherwise, zeros may be appended to reach the precision. For a canonical representation of the value, use `BigDecimal.toString()`.

All flags defined for Byte, Short, Integer, and Long apply.

If the `'#'` flag is given, then the decimal separator will always be present.

The default behavior when no flags are given is the same as for Float and Double.

The specification of width and precision is the same as defined for Float and Double.

52.4.4 Date/Time

This conversion may be applied to `long`, `Long`, `Calendar`, `Date` and `TemporalAccessor`.

`'t'`	`'\u0074'`	Prefix for date and time conversion characters.
`'T'`	`'\u0054'`	The upper-case variant of `'t'`.

The following date and time conversion character suffixes are defined for the `'t'` and `'T'` conversions. The types are similar to but not completely identical to those defined by GNU `date` and POSIX `strftime(3c)`. Additional conversion types are provided to access Java-specific functionality (e.g. `'L'` for milliseconds within the second).

The following conversion characters are used for formatting times:

`'H'`	`'\u0048'`	Hour of the day for the 24-hour clock, formatted as two digits with a leading zero as necessary i.e. `00 - 23`. `00` corresponds to midnight.
`'I'`	`'\u0049'`	Hour for the 12-hour clock, formatted as two digits with a leading zero as necessary, i.e. `01 - 12`. `01` corresponds to one o'clock (either morning or afternoon).
`'k'`	`'\u006b'`	Hour of the day for the 24-hour clock, i.e. `0 - 23`. `0` corresponds to midnight.
`'l'`	`'\u006c'`	Hour for the 12-hour clock, i.e. `1 - 12`. `1` corresponds to one o'clock (either morning or afternoon).
`'M'`	`'\u004d'`	Minute within the hour formatted as two digits with a leading zero as necessary, i.e. `00 - 59`.
`'S'`	`'\u0053'`	Seconds within the minute, formatted as two digits with a leading zero as necessary, i.e. `00 - 60` ("`60`" is a special value required to support leap seconds).
`'L'`	`'\u004c'`	Millisecond within the second formatted as three digits with leading zeros as necessary, i.e. `000 - 999`.
`'N'`	`'\u004e'`	Nanosecond within the second, formatted as nine digits with leading zeros as necessary, i.e. `000000000 - 999999999`. The precision of this value is limited by the resolution of the underlying operating system or hardware.
`'p'`	`'\u0070'`	Locale-specific morning or afternoon marker in lower case, e.g."`am`" or "`pm`". Use of the conversion prefix `'T'` forces this output to upper case. (Note that `'p'` produces lower-case output. This is different from GNU `date` and POSIX `strftime(3c)` which produce upper-case output.)
`'z'`	`'\u007a'`	RFC 822 style numeric time zone offset from GMT, e.g. `-0800`. This value will be adjusted as necessary for Daylight Saving Time. For `long`, `Long`, and `Date` the time zone used is the default time zone for this instance of the Java virtual machine.
`'Z'`	`'\u005a'`	A string representing the abbreviation for the time zone. This value will be adjusted as necessary for Daylight Saving Time. For `long`, `Long`, and `Date` the time zone used is the default time zone for this instance of the Java virtual

		machine. The Formatter's locale will supersede the locale of the argument (if any).
`'s'`	`'\u0073'`	Seconds since the beginning of the epoch starting at 1 January 1970 `00:00:00` UTC, i.e. `Long.MIN_VALUE/1000` to `Long.MAX_VALUE/1000`.
`'Q'`	`'\u004f'`	Milliseconds since the beginning of the epoch starting at 1 January 1970 `00:00:00` UTC, i.e. `Long.MIN_VALUE` to `Long.MAX_VALUE`. The precision of this value is limited by the resolution of the underlying operating system or hardware.

The following conversion characters are used for formatting dates:

`'B'`	`'\u0042'`	Locale-specific full month name, e.g. `"January"`, `"February"`.
`'b'`	`'\u0062'`	Locale-specific abbreviated month name, e.g. `"Jan"`, `"Feb"`.
`'h'`	`'\u0068'`	Same as `'b'`.
`'A'`	`'\u0041'`	Locale-specific full name of the day of the week, e.g. `"Sunday"`, `"Monday"`.
`'a'`	`'\u0061'`	Locale-specific short name of the day of the week, e.g. `"Sun"`, `"Mon"`.
`'C'`	`'\u0043'`	Four-digit year divided by `100`, formatted as two digits with leading zero as necessary, i.e. `00 - 99`.
`'Y'`	`'\u0059'`	Year, formatted to at least four digits with leading zeros as necessary, e.g. `0092` equals `92` CE for the Gregorian calendar.
`'y'`	`'\u0079'`	Last two digits of the year, formatted with leading zeros as necessary, i.e. `00 - 99`.
`'j'`	`'\u006a'`	Day of year, formatted as three digits with leading zeros as necessary, e.g. `001 - 366` for the Gregorian calendar. `001` corresponds to the first day of the year.
`'m'`	`'\u006d'`	Month, formatted as two digits with leading zeros as necessary, i.e. `01 - 13`, where "`01`" is the first month of the year and ("`13`" is a special value required to support lunar calendars).
`'d'`	`'\u0064'`	Day of month, formatted as two digits with leading zeros as necessary, i.e. `01 - 31`, where "`01`" is the first day of the month.
`'e'`	`'\u0065'`	Day of month, formatted as two digits, i.e. `1 - 31` where "`1`" is the first day of the month.

The following conversion characters are used for formatting common date/time compositions.

`'R'`	`'\u0052'`	Time formatted for the 24-hour clock as `"%tH:%tM"`.
`'T'`	`'\u0054'`	Time formatted for the 24-hour clock as `"%tH:%tM:%tS"`.
`'r'`	`'\u0072'`	Time formatted for the 12-hour clock as `"%tI:%tM:%tS %Tp"`. The location of

		the morning or afternoon marker (`'%Tp'`) may be locale-dependent.
`'D'`	`'\u0044'`	Date formatted as `"%tm/%td/%ty"`.
`'F'`	`'\u0046'`	ISO 8601 complete date formatted as `"%tY-%tm-%td"`.
`'c'`	`'\u0063'`	Date and time formatted as `"%ta %tb %td %tT %tZ %tY"`, e.g. `"Sun Jul 20 16:17:00 EDT 1969"`.

The `'-'` flag defined for General conversions applies. If the `'#'` flag is given, then a `FormatFlagsConversionMismatchException` will be thrown.

The width is the minimum number of characters to be written to the output. If the length of the converted value is less than the `width` then the output will be padded by spaces (`'\u0020'`) until the total number of characters equals width. The padding is on the left by default. If the `'-'` flag is given then the padding will be on the right. If width is not specified then there is no minimum.

The precision is not applicable. If the precision is specified then an `IllegalFormatPrecisionException` will be thrown.

52.4.5 Percent

The conversion does not correspond to any argument.

`'%'`	The result is a literal `'%'` (`'\u0025'`). The width is the minimum number of characters to be written to the output including the `'%'`. If the length of the converted value is less than the `width` then the output will be padded by spaces (`'\u0020'`) until the total number of characters equals width. The padding is on the left. If width is not specified then just the `'%'` is output. The `'-'` flag defined for General conversions applies. If any other flags are provided, then a `FormatFlagsConversionMismatchException` will be thrown. The precision is not applicable. If the precision is specified an `IllegalFormatPrecisionException` will be thrown.

52.4.6 Line Separator

The conversion does not correspond to any argument.

`'n'`	The platform-specific line separator as returned by `System.getProperty("line.separator")`.

Flags, width, and precision are not applicable. If any are provided an `IllegalFormatFlagsException`, `IllegalFormatWidthException`, and `IllegalFormatPrecisionException`, respectively will be thrown.

52.4.7 Argument Index

Format specifiers can reference arguments in three ways:

- *Explicit indexing* is used when the format specifier contains an argument index. The argument index is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by "`1$`", the second by "`2$`", etc. An argument may be referenced more than once.

For example:

```
formatter.format("%4$s %3$s %2$s %1$s %4$s %3$s %2$s %1$s",
                    "a", "b", "c", "d")
// -> "d c b a d c b a"
```

- *Relative indexing* is used when the format specifier contains a `'<'` (`'\u003c'`) flag which causes the argument for the previous format specifier to be re-used. If there is no previous argument, then a `MissingFormatArgumentException` is thrown.

```
formatter.format("%s %s %<s %<s", "a", "b", "c", "d")
// -> "a b b b"
// "c" and "d" are ignored because they are not referenced
```

- *Ordinary indexing* is used when the format specifier contains neither an argument index nor a `'<'` flag. Each format specifier which uses ordinary indexing is assigned a sequential implicit index into argument list which is independent of the indices used by explicit or relative indexing.

```
formatter.format("%s %s %s %s", "a", "b", "c", "d")
// -> "a b c d"
```

It is possible to have a format string which uses all forms of indexing, for example:

```
formatter.format("%2$s %s %<s %s", "a", "b", "c", "d")
// -> "b a a b"
// "c" and "d" are ignored because they are not referenced
```

The maximum number of arguments is limited by the maximum dimension of a Java array as defined by *The Java™ Virtual Machine Specification*. If the argument index is does not correspond to an available argument, then a `MissingFormatArgumentException` is thrown.

If there are more arguments than format specifiers, the extra arguments are ignored.

Unless otherwise specified, passing a `null` argument to any method or constructor in this class will cause a `NullPointerException` to be thrown.