debug innovations

Mataki-Lite EMBASIC Reference

Version 2

This document describes how to write scripts in the EMBASIC language for use with Mataki-Lite tags and includes a complete reference for every command, keyword, function and PS variable. Describes features up to and including firmware V1.2.3.

Version History

Version	Date	Changes
1	6 March 2018	First Release for V1.2.2 firmware
2	23 October 2019	Updated for V1.2.3 firmware and EMBASIC V1.4

Related Documents

Mataki-Lite User Guide
Mataki Support Board User Guide

Contents

1. Introduction

This reference guide describes the features in V1.2.3 of the Mataki-Lite firmware. Later firmware versions are likely to support everything in this version. The firmware version is identified in the first line of the text output on PuTTY. It can also be found by typing 'ver' at the prompt.

EMBASIC is a small implementation of the BASIC language for scripts that direct the operation of Mataki-Lite tags. EMBASIC is similar to other BASIC dialects and the core BASIC language constructs are the same. In addition, EMBASIC has extensions to control the Mataki-Lite hardware. If you are new to BASIC, we recommend you familiarise yourself with the language before going any further.

Due to lack of memory and other restrictions, some BASIC features are not supported in EMBASIC:

- All numeric variables are floating point. There are no integer variables.
- There is limited support for strings and no support for string arrays.
- Numeric arrays are supported though they are restricted in size due to memory constraints. Multi-dimensional arrays are not supported.
- Because all I/O goes through a serial port, there is no support for graphics, sound or file operations.

A full list of supported keywords and functions is given in section 6.

Control of the tag's functions is mainly through special variables called "platform-specific variables" or PS variables for short. PS variables are distinguished from ordinary BASIC variables with an underscore as the first character. For example `_ID` is a PS variable that holds the tag's ID. The ID can be read in any BASIC expression which needs a numeric result e.g. `x = _ID`. The ID can be written by assigning a value to it like any other BASIC variable e.g. `_ID = 5`. Just like BASIC string variables, PS string variables end in a '$', for example `_PLATFORM$` contains the platform name.

Some PS variables cause things to happen e.g. `_VBATT` reads the battery voltage when read and logs the battery voltage when written. Some PS variables are read-only e.g. an attempt to write a new value for _PI will give an error. A complete reference for the PS variables is given in section 5.

Note that some of the time functions have standard BASIC keywords and are implemented as such even though they are driven by the tag's hardware and are affected by GPS time updates - see `CLOCK`, `TIMER`, `TIME$` and `DATE$`.

2. Flash Storage

2.1. Logs

Most of the flash memory is used for log entries. Entries are usually GPS fixes created automatically when tracking but other types of entry are possible. A script can use `_LOG$` to store short strings created by the script. Each log entry also contains a log index (entry number), the time/date and the tag's ID number.

From firmware V1.2.3, log entries can be read out using `_LOG$` and `_LOGNUM`. The following script reads and displays the whole log...

```
10 FOR L = 1 TO _LOGUSED
20    _LOGNUM = L
30    PRINT _LOG$
40 NEXT L
```

Each log entry is a string containing fields in the following order:

- Log index
- Sequence number
- Host ID
- Source ID
- Log type
- Time log entry created
- Log contents (varies depending on log type)

This format is the same as that produced by a Mataki-Classic base station and the same Python script can be used to convert the data to other file formats.

2.2. Settings

Some PS variables represent tag settings. They are stored in the flash memory and are reloaded after a re-boot or power cycle. It is good practice not to repeatedly write values stored in the flash as the flash can wear out after many millions of writes. This is easy to do in a loop writing values over and over again. All PS variables that are stored in the flash are identified as such in section 5.

There are some generic 'setting' PS variables that can be used to retain the state of the script between re-boots or can be used as settings for a standard script running on different tags with different requirements. This avoids the need to load a new script if only slightly different behaviour is needed between tags - see `_S1` to `_S4` and `_S1$` to `_S2$`.

3. Developing and Loading Scripts

EMBASIC scripts can be written using any plain text editor such as Notepad and saved to disk. Files should be given the extension '.bas'.

Script files can be loaded into the Mataki-Lite emulator software on a PC using the `LOAD` command, be saved to disk using the `SAVE` command and files listed using the `FILES` command. The emulator software is a convenient way to develop and test scripts. It isn't identical to the real tag and it is no substitute for real-world testing but it is convenient and fast and has enough hardware emulation to help with code development.

For real hardware, script files can be programmed into the flash on a tag using the load_script utility. They can be edited using the built-in EMBASIC editor on the tag and saved to the flash using the `SAVE` command on the tag. It is also possible to 'paste' a script into PuTTY for testing.

Advanced base stations can also load new scripts over the radio. If you don't want this to happen when you are developing scripts, set `_SCRIPTLOCK` to '1'.

4. Script Examples

4.1. Introduction to GPS

The main purpose of Mataki-Lite is to log the positions of a bird or animal over time. The GPS module calculates the position of the tag from signals received from GPS satellites. It also gets the time from the same source. However, this process is complex and it can take some time, particularly the first position fix because the tag has no idea where it is on the Earth, it doesn't know where the satellites are and it doesn't know what time, date or year it is. This is a simplified description of the process the GPS follows to know where it is...

1. Initially, the GPS receiver scans all the radio channels in turn looking for satellites. This is a random process and it is pure luck how long it takes.
2. When it finds a satellite, it gets the time and week number and starts downloading something called an almanac. The almanac is a mathematical description of the orbits of all the GPS satellites. There is also a fine correction for each satellite called the ephemeris data.
3. Once the almanac is available, the module can work out which satellites are near the one it has found and what channels they are transmitting on. In this way, the module can find multiple satellites and receive multiple sets of data.
4. The full almanac and ephemeris data can take a single satellite 10 minutes to broadcast. However, the data each satellite transmits is out of sync. with the other satellites, so if the receiver can listen to many satellites, the complete data set can be downloaded more quickly. As more and more is known about the satellite orbits, the module can lock on to more satellites and the process speeds up.
5. At some point there is enough information to resolve the week number into a calendar date and we get the GPS time/date from the module. This is shown on the PuTTY text and the tag's on-board clock is set to GPS time. GPS time is precisely the same for all satellites and it is set to UTC time (same as GMT/London).
6. The module gets different times in the transmissions from each satellite due to the time the signal takes to travel from the satellite to the module. However the transmission also contains the satellite's position. With some complex maths, the module can work out the distance to all the satellites and calculate its own position. This requires at least 3 satellite signals but a more accurate position is obtained with at least 4 satellites.

4.2. Getting a GPS Fix

When writing scripts, the complex process of getting a GPS fix is fully automated but minimising power consumption requires that the GPS is only on when it is needed. This is the simplest way to log a GPS fix:

```
1000 REPEAT
1010    _GPS = 1
1020 UNTIL _FIXVALID = 1
1030 _GPS = 0
```

Setting `_GPS` to 1 turns on the GPS module and also runs the GPS firmware built into the tag. That is why it is necessary to loop round constantly setting `_GPS` to 1. When a fix is obtained `_FIXVALID` will be set to 1 and the loop will terminate.

`_GPS = 0` turns the GPS off. It is important to note that the GPS module power is still on, the module is merely sleeping. In this state the module retains the almanac and ephemeris data and its real-time-clock continues to run. When the GPS is turned back on, there is no need to download all the data again, the module knows roughly where it is and it knows the time, so a fix will be obtained quicker.

If you want to turn off the GPS power, set `_GPS` to -1. Once the module has been turned off, getting another fix will require the full almanac and ephemeris to be reloaded. To help with step 1 in section 4.1, the tag firmware will load the current time and last known position into the GPS module to give it a clue which satellites might be available. However, in practice, this doesn't shorten the time to first fix by much.

In the example above, setting `_GPS` to 1 not only turns the GPS module on but also automatically logs a fix when it is obtained. Repeated loops will log multiple fixes (tracking). If `_GPS` is set to 2, the fix will not be logged. This allows a user to post-process the fix to decide whether it meets the application's criteria e.g. it is common to require at least 4 satellites and the DOP values to be low enough. The fix can then be logged by setting `_LOGFIX` to 1.

The latitude, longitude and all the other fix data are available as PS variables, see section 5.3. These can be examined by the script to make decisions e.g. GEO-Fencing an area of interest causing an increase in the logging rate.

The biggest problem with the simple script above is that it has no timeout. So an animal living in a burrow or rolled over onto the tag would never receive a fix and the script above would leave the GPS module on until the battery went flat. Adding a timeout is more complicated than it appears at first as the tag's clock can be set to the GPS time when it is obtained from a satellite. This will cause the tag's time to jump. The supplied example tracker script contains code which can deal with this and still timeout correctly.

`_GPSSIGNAL` can be used to estimate whether the GPS module has any chance of getting a fix or not. For example, for animals that live in burrows, it is worth checking the signal 20-30 seconds after turning on the GPS if no fix has been obtained. If the signal level is low, say less than 30, it is unlikely that a fix will be obtained and the GPS can be turned off to save power. Furthermore, if the animal's behaviour is such that it stays in the burrow for a long time, then the GPS can be left off for a long time.

In summary, the script has a lot of flexibility in operating the GPS but the primary challenge is to reduce the amount of time the GPS module stays on. If the script sees anything that suggests the module can't or won't get a fix, it is better to wait than keep trying.

4.3. Automated Log Downloads

When `_RADIO` is set to 1, a heartbeat is sent to the base station. If the base station is in radio range and is not busy with a different tag, it will reply to the heartbeat and attempt to establish a connection with the tag on a different channel. If contact is made, the base station will then request any log data it doesn't already have from this tag (based on tag ID) and the tag will send the necessary log entries. At the end of the transfer, the base station will instruct the tag to re-boot. This all happens automatically and doesn't need to be scripted.

However, if the tag fails to make contact with the base station or loses contact during the transfer, it will not get re-booted by the base station. In this case your script needs to time out and continue its normal activities (typically this would be tracking). Furthermore, even though the log transfer is automatic, the code that does it needs processing time to execute, so the script must loop during the transfer. The following example illustrates this:

```
2000   _RADIO = 1
2010 REM 20s is long enough
2011 REM If contact is made, we will be re-booted
2020   FOR d = 1 to 2000
2030     DELAY 0.01
2040   NEXT
2050   _RADIO = 0
```

`_RADIO = 1` starts the process by sending a heartbeat to the base station. The loop from line 2020 to line 2040 executes 2000 times, each time with a 0.01 second (10ms) delay on line 2030. The total time to execute the loop is 20 seconds, after which `_RADIO = 0` will turn off the radio. If contact is made with the base station, it is likely the tag will be re-booted during the loop, if not the radio will be turned off after 20 seconds.

An alternative, better method is to use the `_CONTACT` variable to control the timeout (see description of `_CONTACT` in section 5.4):

```
2000 _RADIO = 1
2010 REPEAT
2020 UNTIL _CONTACT = 0
2030 _RADIO = 0
```

Hybrids of these two approaches allow both the base station and tag to control the timeout.

On some base station types, it is possible to upload a new script to the tag once the logs are sent. Before this happens, the current script is stopped and the whole of the script upload process is handled by the tag's firmware. If the whole script arrives at the tag intact, the new script is programmed into the tag's flash memory and the tag is re-booted, causing it to load and run the new script. If the new script is not successfully uploaded e.g. because the tag goes out of radio range during the transfer, the firmware will re-boot the tag and the old script will be used again. On next contact, the base station will attempt to upload the new script again. This continues until the new script is successfully uploaded.

IMPORTANT: You must set the script to start automatically (Auto-run ON) when loaded onto the tag, or the script will stop working after the first successful log transfer.

In summary, there are 3 cases that your script needs to cater for:

1. The base station is in range. The firmware will upload logs and (optionally) receive a new script. In this case, your script loop will never end because the tag will be re-booted.

2. The base station is out of range. In this case, there is no point in leaving the radio on and the `_CONTACT` variable allows your script to terminate the loop early.

3. The base station is initially in range and the log transfer starts, then the tag goes out of range of the base station. In this case, your loop will terminate when `_CONTACT` gets to zero or you may choose to implement your own timeout.

4.4. Sending and Receiving Private Messages

Mataki-Lite supports sending and receiving private radio messages between tags (as opposed to sending messages to the base station). Private messages are very simple to use and can be used for all kinds of purposes. To keep them from interfering with other tag's log messages, private messages should not use channels 16 and 20 (see `_RADCHAN` in section 5.4 for more details). In this context, 'Private' simply means proprietary non-standard messages. They are not encrypted and not directed to any specific tag. Anyone with a suitable radio or Mataki tag can receive all private messages.

The following example transmits a "Hello" message every 2 seconds:

```
10 _RADCHAN = 29
20 _RADTXPWR = 10
30 _RADSPEED = 2
40 _RADIO = 2
50 _RADMSG$ = "Hello"
60 DELAY 2
70 GOTO 50
```

To enter private message mode `_RADIO` is set to 2 in line 40, then setting `_RADMSG$` to a string in line 50 will cause the message to be transmitted using the settings in lines 10-30. A message can be up to 48 characters. If you want to put a number in the message, use one of the standard BASIC conversion functions e.g. `STR$()` to convert it to a string first. If you want a message to go to a specific tag, the sending script must add some address information to the message e.g. the tag's ID and your receiving script must look at the messages to decide whether that message is to be used or discarded.

The following example receives private messages:

```
10 _RADCHAN = 29
20 _RADSPEED = 2
30 _RADIO = 2
40 REPEAT
50   r$ = _RADMSG$
60 UNTIL r$ <> ""
70 PRINT r$
80 GOTO 40
```

Reading `_RADMSG$` in line 50 copies the message to `r$` and clears out the message so the next time `_RADMSG$` is read it gives an empty string. This way a message is printed on line 70 for each message received. However, this simple system relies on the transmitting end not sending multiple messages close together as only one message can be held in `_RADMSG$` at a time. If `_DBGRADIO` = 1, received data will be printed along with signal strength and frequency offset information to help develop scripts with private messages.

5. Platform Specific Variables

5.1. Constants

_PI	Pi.
_RADTODEG	Multiply by this to convert Radians to Degrees. **Example**: `10 PRINT _PI * _RADTODEG` Prints "180".
_DEGTORAD	Multiply by this to convert Degrees to Radians. **Example**: `10 PRINT 180 * _DEGTORAD` Prints "3.14159".
_EARTHRM	Radius of the Earth in metres.
_PCBVER	The PCB hardware version.
_VER$	The base firmware version as a string, equivalent to the `VER` command.
_PLATFORM$	The platform name: "Mataki-Lite"

5.2. Settings

_ID **Access: RWF**	The tag ID.
_STDLY **Access: RWF**	Start up delay in minutes. Setting this to a non-zero value will put the tag into its lowest power state for the specified time when it is powered on. After the startup delay, the tag runs the script as normal. The startup delay doesn't occur after a soft reset as might happen when the base station resets the tag. A startup delay can be used to get a batch of tags ready in advance of deployment, if deploying the tags is a time consuming or regulated process e.g. accessing an island at high tide. During the startup delay the tag consumes a negligible amount of power, a small 100mAh battery would last more than 1 year.

_S1...S4
 	
Access: RWF
	These 4 values have no 'meaning' to the tag - they can be used to store any user value between resets. They are stored as 32-bit integers with a range of approx. -2 billion to +2 billion.
	
	Example:
	
	Imagine issuing the following commands in immediate mode:
	
	`> x = 5`
	`> _S1 = 10`
	
	When recalled, these would have the values given.
	
	Now, imagine resetting the device and what would happen when recalling these values:
	
	`> ?x`
	`*** Error: Uninitialised variable`
	
	`> ?_S1`
	`10`
	
	The value for `_S1` has survived the reset of the device because it has been saved in flash memory.

_S1$...S2$

Access: RWF

String equivalent to `_S1..._S2`. Only the first 32 characters are stored.

See the example above and substitute the numeric values for strings to see how this can be used.

_OWNER$

Access: RWF

The owner of the device. This string is sent to the base station to help identify the tag. Only the first 32 characters are stored.

_STUDY$

Access: RWF

Can be used to identify an experiment group. This string is sent to the base station to help identify the tag. Only the first 32 characters are stored in the flash.

_SCRIPTLOCK

Access: RWF

Locks and unlocks the script. When the script is locked, it cannot be overwritten by the base station. This setting does not affect the flash loader. Locking the script is useful when developing new scripts to stop it being overwritten by the base station. 0 = Unlocked, 1 = Locked.

Access: **R** – Value can be read **W** – Value can be written to
F – Value is stored in the flash memory between power cycles

5.3. GPS Control

_GPS

Access: RW

Reading `_GPS` tells you whether the GPS power is on, 1 = on, -1 or 0 = off.

The value written to `_GPS` controls the mode of operation. The following modes are supported:

- -1 : Turns the GPS 1.8V power off. This is the lowest power mode but the GPS cannot be used. Normally the GPS is switched on and off with modes 0 and 1 (or 2). The first time the tag is powered on, the GPS will be in mode -1. If a startup delay is specified, it will happen at the minimum current. Once the GPS is set to mode 0, 1 or 2, a tag reset (such as may be triggered by the base station) will put the GPS back to mode 0 to speed up re-acquisition of a fix.
- 0 : Turns the GPS off (to save power). In mode 0, the GPS 1.8V power is on but the GPS module is in its lowest power state. Crucially, the GPS module retains the satellite almanac and ephemeris data and runs the real-time clock in mode 0.
- 1 : Turns the GPS on and tries to get the GPS time and a position fix. It is important to continue to set `_GPS` to 1 in a loop to give the GPS time to get a fix. If a fix is obtained, `_FIXVALID` will be set to 1 and the `_FIX` variables will contain the current position. The fix will also be added to the log. Continuing to loop setting `_GPS` to 1 will log more positions (tracking).
- 2 : Same as 1 but does not log positions. This mode can be used to obtain a position fix and make decisions about whether to carry out further actions based on location e.g. for GEO-Fencing, or the positions can be processed before logging e.g. averaged or positions discarded.

_GPSSIGNAL

Access: R

A value which loosely represents the signal strength received by the GPS module. This is calculated by adding together the SNR values of the satellites in view. Before the GSV message is received, this information is unknown, so _GPSSIGNAL has a value of -1.

See section 4.2 for an explanation of how this variable might be used to save power.

_FIXLAT
_FIXLON
_FIXALT

Access: RW

The latest GPS fix latitude, longitude and altitude.

Whenever `_GPS` is written, these variables will be overwritten with the latest information and can be read by the script. Take care not to use the values unless `_FIXVALID` is 1.

If these variables are written by the script, setting `_LOGFIX` will create a GPS Fix log entry using the values written. In this way, a script can read the GPS position then process the values e.g. averaging them, then write a processed result to the log.

_FIXPDOP
_FIXHDOP
_FIXVDOP

Access: RW

The latest GPS fix 'Dilution of Precision' values (a measure of fix uncertainty).

These variables can be written and behave the same way as `_FIXLAT` above.

_FIXSATS

Access: RW

The number of satellites used in the latest GPS fix.

This variable can be written and behaves the same way as `_FIXLAT` above.

_FIXVALID

Access: R

Reading this variable tells you whether the `_FIX` variables above are valid, 1=valid, 0=not. To obtain a fix, set `_GPS` to 1 or 2 in a loop until `_FIXVALID` is 1.

It is up to the user script to decide whether the fix has a high enough 'quality' to meet the application requirements. For example, it is common to require 4 satellites and low DOP values before considering a fix 'good enough'.

To apply filters such as these, simply wait until `_FIXVALID` is 1 then check the `_FIX` variables. If the fix isn't good enough, loop round until it meets your requirements. Note that in mode 1, each loop will add an entry to the log, so for this type of script, use `_GPS` mode 2, then use `_LOGFIX` to store the fix in the log.

_LOGFIX

Access: RW

Reading this variable does nothing and always returns 0.

Setting this to a non-zero value will cause the `_FIX` variables to be written to the log as a FIX log type with a fix quality of the value written. This value can be used to indicate that the fix was created by the script and may bear no resemblance to the actual position, or different values can be used for different algorithms etc. Fix quality is stored as 8 bits. The value for a normal GPS fix will always be less than 10 so we suggest using values 10-255.

_DBGGPS

Access: RW

When set to 1, displays extra debug information when the GPS is on. All the NMEA messages are displayed plus decoded information.

Access: **R** – Value can be read **W** – Value can be written to
F – Value is stored in the flash memory between power cycles

5.4. Radio Control

_RADIO

Access: RW

Reading `_RADIO` tells you whether the radio power is on 1 = on, 0 = off.

Permissible write values and their actions are:

- 0 : Turns the radio off (to save power)
- 1 : Turns the radio on and tries to contact the base station to download the log. This mode is completely automatic; however the script needs to loop to allow the processing to happen (see section 4.3).
- 2 : Puts the radio into private message mode (see `_RADMSG$` below). This mode is an extremely simple way of sending short messages between tags for testing or other purposes. To exit private message mode, turn the radio off.
- 100 : Generates a continuous carrier on the selected channel at the selected transmit power until the radio is turned off. This mode is intended for testing and shouldn't be used in tracking scripts. However it is possible to generate short CW 'pips' which could be used with an SSB radio as an emergency locator.

_RADCHAN

Access: RW

The channel to use in modes 2 and 100. Assuming `_FREQBASE` is set to 868MHz, channel 0 will be centred on 868MHz. The channel spacing is 50kHz, so channel 1 is at 868.050MHz, channel 2 is at 868.100MHz and so on. There are 40 channels (0-39) over a 2MHz band (868-870MHz). The same principle applies to all `_FREQBASE` settings. The centre frequency (F_c) in MHz can be calculated from the channel number (c) as follows:

$$F_c = \_FREQBASE + 0.05c$$

`_RADCHAN` is used when `_RADIO` is set, so it needs to be set up before changing mode and the radio needs to be turned off to change channel.

To ensure reliable data transfer and avoid interference between channels and to other users, it is important to choose channels carefully. Firstly, local regulations must be adhered to - these stipulate the allowable frequencies and power levels for your country. Note that in most countries, it is a criminal offence to radiate outside the allowed band even if your centre frequency is inside the band i.e. your transmission is wide enough to go outside the band.

The data rate (see `_RADSPEED` below) will determine the spread of frequencies that the transmitter will generate (known as the bandwidth of the transmission). For data rate 1 the bandwidth is less than 50kHz, so each channel can be used separately from the others. For data rate 2, the bandwidth takes up 2 channels, so you need to keep these transmissions at least 2 channels apart and 2 channels from anything else you need to avoid like the Mataki base station channels 16 (868.8MHz) and 20 (869.0MHz).

Channel 0 should not be used except for carrier testing (mode 100) as half of the transmission's bandwidth will be below 868MHz (and the other half above), which is outside the band and thus a breach of the regulations. Likewise, channels 1 and 39 cannot be used at data rate 2 as the wider bandwidth will result in signals below 868MHz and above 870MHz respectively.

In the UK, the first 600kHz of the band is used by LoRa devices and is likely to get more congested with the IOT rollout, the last 300kHz has a 5mW power limit and there are 4 segments of the band which are reserved for radio alarms. Taking into account all the restrictions mentioned above, the recommended private message channels for the UK are 29 (869.45MHz) and 31 (869.55MHz).

Similar restrictions will apply to other countries. If you only use `_RADIO` mode 1 and the correct `_FREQBASE` is used, the radio will automatically operate within the regulations.

_RADTXPWR

Access: RW

The transmit power to use in modes 2 and 100. The only permissible values are -30 -20 -10 0 10 12 (all in dBm). Other values will generate errors or have no effect. In general, higher transmit powers result in longer range but also use more battery power. Optimal systems adjust the transmit power to maintain a connection using the minimum power.

This variable is used when `_RADIO` is set, so it needs to be set up before changing mode and the radio needs to be turned off to change the power setting. The default setting is +10dBm.

_RADSPEED

Access: RW

The radio data rate. In general, lower data rates will give greater receive sensitivity resulting in greater radio range. This variable is used when `_RADIO` is set, so it needs to be set up before changing mode and the radio needs to be turned off to change the data rate.

Permissible write values and their actions are:

- 1 : Set rate to 1200 baud, GFSK modulation
- 2 : Set rate to 38400 baud, GFSK modulation

_RADMSG$

Access: RW

Message to send or receive in mode 2 (private message mode). Messages are limited to 48 characters of text, though there is no limit to the number or content of messages that can be sent. All tags in mode 2 (in radio range) will receive the message.

Note that the message is not encrypted. If you want to send secure messages that other receivers cannot decode, you must encrypt the data in the script before sending it. See section 4.4 for more information.

_RADRSSI

Access: R

The received signal strength for the last private message received in dBm.

_DBGRADIO

Access: RW

When set to 1, displays extra debug information during radio activity.

_FREQBASE

Access: RWF

This sets the base (channel 0) frequency for the radio in MHz. Note that this is not related to the 'base' station in any way. It is simply the radio frequency the tag uses for channel 0. In private message mode (`_RADIO` = 2) the channel can be set using `_RADCHAN`. In automated mode (`_RADIO` = 1) the tag firmware is in control of the radio channel but still uses the `_FREQBASE` setting. So it is critical that `_FREQBASE` is set correctly or the tag will not be able to contact the base station. By default, tags are set to 868MHz. Users are responsible for setting a suitable frequency for the country of use. We recommend the following settings:

Europe	: 868MHz (default)
USA	: 916MHz

The frequency can be checked by reading this variable at the BASIC prompt by typing `?_FREQBASE` and can be set by typing `_FREQBASE=868.2` for example.

Whatever the base frequency setting, the tag may use radio channels 0-39 which represent a 2MHz band e.g. if `_FREQBASE` is 868MHz, the tag may transmit and receive signals between 868MHz and 870MHz. This has to be taken into consideration when setting `_FREQBASE`.

Changes in this value require a re-boot to take effect. Consequently `_FREQBASE` cannot be used to change radio frequency during operation. It is designed to set the operating frequency for the life of the tag, based on the country of use.

_FREQCAL

Access: RWF

This is a factory calibration value for the radio. It is the offset for the base frequency in Hz with a range of -25kHz to 25kHz. We recommend you don't change this value unless you have the equipment to measure and set it again. Changes in this value require a re-boot to take effect.

_CONTACT

Access: R

In automated mode (`_RADIO` = 1), this read-only value says how many seconds are left before the tag should timeout on its radio operations. It is initially set to a small value when the radio is turned on and then set to a value given by the base station any time there is radio activity.

When there is no radio activity, the value counts down each second until it reaches zero, at which point the script should turn off the radio to save power.

By following this algorithm, tags which are out of radio range of the base station will only turn on the radio for a few seconds before giving up and tags which lose contact while communicating with the base station also timeout quicker than they would if a fixed timeout was used. This saves battery power.

Access: **R** – Value can be read **W** – Value can be written to
F – Value is stored in the flash memory between power cycles

5.5. Log Control

_LOGCLEAR **Access: RW**	Set to 1 to clear the log. This also sets `_LOGNUM` to 0.
_LOGUSED **Access: R**	The number of log entries used
_LOGCAP **Access: R**	The total log capacity
_LOGFIX **Access: RW**	See section 5.3 GPS Control.
_LOG$ **Access: RW**	Writing a string to this variable causes a log entry to be written with that text. There is only enough space in a log entry for 18 characters. This also causes `_LOGNUM` to be set to the new value of `_LOGUSED` (the last log entry). Reading this variable returns a string containing the log entry at the index given by `_LOGNUM`. This is not the same as the string written with `_LOG$`. If `_LOGNUM` is not a valid entry number, then the returned string is empty. See section 2.1 for further information.
_LOGNUM **Access: RW**	Determines which log entry is returned when `_LOG$` is read. Valid values are 1 to `_LOGUSED`. On start up this points to the last log entry.

Access: **R** – Value can be read **W** – Value can be written to
F – Value is stored in the flash memory between power cycles

Analogue Inputs

_LIGHT **Access: RW**	Reading this variable causes a light reading to be taken. The value is an ADC reading in the range 0-4095. The meaning of the reading is application dependant but it can be used to detect dawn/dusk or an animal in a burrow. Writing 1 to this variable causes a light reading to be logged.
_VBATT **Access: RW**	Reading this variable causes a battery voltage reading to be taken. The value is in Volts. Writing 1 to this variable causes a battery voltage reading to be logged. The battery voltage will be dependent to some extent on the load, so whether the GPS or radio are on is likely to affect the reading.
_SEASENSE **Access: R**	Reading this variable causes a sea sense reading to be taken. The value is an ADC reading in the range 0-4095. This is connected to a pin on the tag's edge connector and pulled up to 3.3V with a 30k resistor. If the edge connector is exposed and dipped in sea water, the ADC reading will go down. In this way, a tag can detect a bird diving in the sea. Salt residue will tend to keep the value lower for a while, so we recommend experimenting with this feature to find a suitable threshold if you plan to use it.

Access: **R** – Value can be read **W** – Value can be written to
F – Value is stored in the flash memory between power cycles

5.6. Time and Date

_UPTIME **Access: R**	The time since reset in seconds.

_SLEEP
 	
Access: RW
	Reading this variable gives the time spent sleeping in seconds.
	
	Writing to this variable puts the tag in a low power state for the number of seconds specified. Sleep times are rounded down to the nearest second (the part after the decimal point is ignored).
	
	The tag should be put into a sleep state whenever it is inactive to save battery power. Before going to sleep, turn off the GPS and radio, see `_GPS` and `_RADIO`.
	
	In sleep mode, the CPU stops running the script and messages from the GPS and radio are ignored. Sleep terminates when the sleep time has passed or the user presses a key on the PuTTY terminal. The key press itself is discarded so, if you want to get a key input from sleep mode, you need to press the key twice and read the key after the `_SLEEP` as in the following example:
	
	`10 REPEAT`
	`20   _SLEEP = 10`
	`30   a$ = INKEY$`
	`40 UNTIL a$ = "C"`

Access: **R** – Value can be read **W** – Value can be written to
F – Value is stored in the flash memory between power cycles

See also the EMBASIC keywords `CLOCK`, `DELAY`, `TIMER`, `TIME$` and `DATE$`.

5.7. Other Controls

_RESET **Access: RW**	When read, gives the reason for the last reset. Valid reasons are: • 0 : Unknown (a normal power up) • 1 : Watchdog triggered • 2 : Reset request (`_RESET=1` or base station) Writing 1 to this variable resets the device.
_LED **Access: RW**	Controls the User LED (red), 0 = Off, 1 = On.
_SUPPLEDS **Access: RW**	Controls the Debug LEDs on the support board, 8 bits, one for each LED. Note that these LEDs cannot be used at the same time as `_TP3`/`_TP4`.
_SUPPLED **Access: RW**	Controls a user LED on the support board, 0 = Off, 1 = On.
_SUPPCONN **Access: R**	Tells you whether the tag is plugged into a support board, 1 = Inserted, 0 = Not.

_TP1/_TP2

Access: RW

Reading `_TP1` or `_TP2` returns the current output state.

Writing 1 or 0 to these variable sets the test points 1 and 2 high or low.

For minimum power consumption test points should be high. This is important if your deployed script uses the test points as it will affect the battery life.

_TP3/_TP4

Access: RW

Similar to `_TP1`/`_TP2` except that the test points are on the support board. These pins have test point loops which can easily be connected to a scope probe. These pins are shared with the Debug LEDs and `_SUPPLEDS` cannot be used at the same time.

Access: **R** – Value can be read **W** – Value can be written to
F – Value is stored in the flash memory between power cycles

If test points are used to control other electronics, please note that test points have 10k pull-up resistors on them, so they consume extra power when low. The electronics should be designed to be driven by low pulse signals which stay high most of the time for lowest power consumption. Under no circumstances should the test points be externally driven (used as inputs).

In addition to the obvious requirement not to connect 2 outputs together, the CPU reads the state of the test points at reset and both pins must be high or the CPU will not boot. It can be difficult to meet this requirement if the external circuitry is powered off when the CPU boots as ESD protection diodes inside the (powered off) device will conduct and stop the signals being pulled up to 3.3V. Robust solutions are possible but will likely require discrete transistors or special devices. Please contact us if you need advice.

6. EMBASIC Reference

6.1. Keywords

ABS

Returns the absolute value of a numeric value.

Syntax:

```
ABS(x)
```

Example:

```
10 PRINT ABS(10)
20 PRINT ABS(-5)
```

Prints "10" and "5".

ASC

Returns the ASCII code for the first character of a string.

See also `CHR$` which does the opposite conversion.

Syntax:

```
ASC(x$)
```

Example:

```
10 PRINT ASC("Hello")
```

Prints "72" (72 is the ASCII code for 'H').

ATN

Returns the trigonometric arctangent of a numeric value in radians.

See also `TAN`, `SIN` and `COS`.

Syntax:

```
ATN(x)
```

`x` is a value in radians.

To convert the result to degrees, multiply by `_RADTODEG`.

Example:

```
10 x = ATN(10)
20 PRINT "Arctan 10 (rad): " ; x
30 x = x * _RADTODEG
40 PRINT "Arctan 10 (deg): " ; x
```

Prints "Arctan 10 (rad): 1.47113" and "Arctan 10 (deg): 82.2894".

BEEP

If a PuTTY terminal is connected, then a short beep is played. Otherwise, has no effect.

Example:

```
10 IF x > 10 THEN BEEP
```

Terminal beeps when x is larger than 10.

BREAK

Exits a `FOR..NEXT` or `REPEAT..UNTIL` loop early.

Not legal in immediate mode.

The end of the loop must be further down the program (at a higher line number). Will give error "Can't BREAK out of loop" if the end of the loop cannot be found.

Example:

```
10 FOR i = 1 TO 10
20   IF i > 5 THEN BREAK
30   PRINT i
40 NEXT
```

Prints the values 1 to 5.

CHR$

Converts an ASCII code to its equivalent character.

Returns '?' if character is not in the range 0-255.

See also `ASC` which does the opposite conversion.

Example:

```
10 PRINT CHR$(72)
```

Prints the character 'H'.

CLOCK

Returns the whole number of seconds since midnight on 1 January 1970. This is a read-only value.

All time/date values are in UTC because the time is obtained from the GPS satellites.

`CLOCK` is useful for measuring elapsed time. Unlike `TIMER` which is also supported and is a common feature of other BASIC variants, `CLOCK` doesn't reset to zero at midnight, so events that span midnight can be safely timed simply by taking an initial copy of `CLOCK` at the start, one at the end and subtracting to get the duration in seconds.

Note that, in common with all the other timer functions, if the GPS time has not been obtained e.g. on first switch on, then the time will be wrong and scripts should be written to expect this and anticipate a sudden time jump when the time is obtained.

Example:

```
10 t1 = CLOCK
20 DELAY 20
30 t2 = CLOCK
40 PRINT "Duration: "; t2 - t1; "seconds"
```

CLS

Clears all text from the PuTTY terminal display.

Example:

```
10 CLS
```

COLOR

Used with `PRINT` to change the text colour.

The supported values and their colours are:

0	Default (Grey)
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White

Example:

```
10 PRINT COLOR(1) ; "Red" ; COLOR(0);
   "Normal"
```

Prints “Red” in red, and “Normal” in the default colour.

CONT

Resumes a program after it has been stopped by the escape key or with `STOP`.

Not legal while running.

Example:

The following program prints “Hello” then stops.

```
10 PRINT "Hello"
20 STOP
30 PRINT "Goodbye"
```

When we run it, it produces the following output...

```
> RUN
Hello
STOP at line 20
```

Then we can continue from where it was left off by typing `CONT` as follows...

```
> CONT
Goodbye
```

COS

Returns the trigonometric cosine of a numeric value in radians.

See also `ATN`, `SIN` and `TAN`.

Syntax:

```
COS(x)
```

`x` is a value in radians.

To convert the return value to degrees, multiply by `_RADTODEG`.

Example:

```
10 PRINT COS(0)
20 PRINT COS(_PI / 2)
30 PRINT COS(_PI)
40 PRINT COS(3 * _PI / 2)
```

Prints “1”, “0”, “-1”, and “0”.

DATA

Stores numeric and string constants that can be accessed by the `READ` command.

Syntax:

```
DATA [const1][,const2][,const3]...
```

`DATA` statements are read in the order they occur starting with the statement at the lowest line number. Each value is read in the order given until all the values have been read, then the next `DATA` statement is located and reading continues. The `RESTORE` command can be used to reset the read order.

Any number of `DATA` statements can be placed anywhere in a program. The values must be constants and can be of numeric or string type. The type being read must match the variable type in the `READ` statement or a "Type Mismatch" error occurs.

It is good practice to always use quotes around strings in `DATA` statements. Quotes are only strictly required if the string a) contains commas b) contains leading or trailing spaces that are required to be read or c) contains BASIC keywords.

The `READ` command can come before or after the `DATA` it is reading. If the program runs through a `DATA` section it is treated the same as `REM` i.e. the whole line is ignored/skipped.

Has no effect in immediate mode.

Examples:

```
10 DATA "Apple", "Banana", "Cherry"
20 READ A$, B$, C$
30 PRINT A$, B$, C$
```

Prints the types of fruit stored by `DATA`

```
10 READ N$, H
20 DATA "Alice", 158
30 PRINT N$ ; " - " ; H ; "cm"
```

An example where `READ` has been used before `DATA` and multiple data types have been used.

Prints "Alice - 158cm"

DATE$

Returns a string representation of the date in the format "DD-MM-YYYY". This is a read-only value.

All time/date values are in UTC because the time is obtained from the GPS satellites.

See also `TIME$`.

Example:

```
10 PRINT DATE$
```

DELAY

Waits for the given number of seconds. The value can be given in fractions of a second.

On the emulator (Windows) the actual delay is accurate to around 20ms. On a Mataki-Lite tag the delay is accurate to around 2ms.

`DELAY` does not depend on the current time, so it is safe to use `DELAY` while the GPS is on. Time corrections obtained from the GPS will not affect the duration of `DELAY`.

During the delay, Mataki-Lite consumes approx. 4.2mA (with the GPS and radio turned off) so it is not recommended that scripts have long delays. For long delays, use `_SLEEP`.

NOTE: When delayed, some system routines will not be serviced. For this reason it's better to have many small delays rather than one large one.

See also `_UPTIME`, `CLOCK`, `TIMER`, `TIME$` and `DATE$`.

Example:

```
10 PRINT "Hello"
20 DELAY 0.5
30 GOTO 10
```

Prints "Hello" every 0.5 seconds.

DIM

Declares an array variable and allocates memory for it.

Syntax:

```
DIM variable(size)
```

The array can be indexed from 0 up to and including `size`.

Gives an "Out of memory" error if too much memory has been dimensioned.

Gives "Redimensioned array" error if an array variable with the same name already exists.

Example:

```
10 DIM x(10)
20 FOR i = 0 TO 10
30   x(i) = i * 2
40 NEXT i
```

Creates an array of length 11 and fills it with the even numbers from 0 to 20.

END

Terminates the program and returns to the prompt. This is optional at the end of the script (highest line number), as EMBASIC exits automatically when reaching the end of the script. However, it is more normal to have subroutines at the end of a script, so `END` is needed between the main program and the start of the subroutines.

Not legal in immediate mode.

It is strongly recommended that this is only used in testing and not in any deployed script because going back to the prompt while running means the tag will never manage to start the script again, effectively leaving the tag completely idle until it runs out of battery. All deployed scripts should loop forever.

Example:

```
... <set x to something>
10 IF x > 10 THEN END
20 PRINT "Still going!"
```

Exits early if `x` is greater than 10. Otherwise prints "Still going!".

EXP

Returns the mathematical constant *e* raised to a given exponent.

Syntax:

```
EXP(x)
```

Example:

```
10 PRINT EXP(5)
```

Prints “148.413”.

FILES

Returns files ending “.bas” in current directory. Equivalent to the MSDOS command "DIR *.bas".

Only supported on emulations, not on tags.

Not legal while running.

FIX

Rounds a given numeric value by truncating the non-integer component (rounds toward zero).

`FIX` simply crops off the decimal point leaving the integer and its sign. For example, -1.6 becomes -1, 12.4 becomes 12.

See also `INT` and `ROUND`.

Examples:

```
10 PRINT FIX(10.257)
```

Prints “10”.

```
10 PRINT FIX(-19.1)
```

Prints “-19”.

FOR

TO
STEP
NEXT

Loops through a block of instructions a given number of times adjusting the value of a variable on each loop.

Syntax:

```
FOR variable = start TO end [STEP amount]
  .
  . <looped instructions>
  .
NEXT [variable][,variable2]...
```

The block of instructions executed on each loop is contained between the `FOR` and `NEXT` statements. The value of *`variable`* changes on each loop.

`start` is the initial value for *`variable`*
`end` is the final value for *`variable`* at which the loop will exit
`amount` is how much to increment *`variable`* by on each loop

`start, end` and *`amount`* can be constants or expressions containing other variables.

If `STEP` is not specified, a default increment of 1 is used. If *`amount`* is negative, the variable will count down and the value of *`end`* must be less than *`start`*.

The most common use of a `FOR..NEXT` loop is to do something a fixed number of times. For example...

```
10 FOR x = 1 TO 10
20   PRINT x
30 NEXT
40 PRINT "Done, x="; x
```

Prints the values 1 to 10, then "Done, x=11"

On the first loop, `x` is 1. When `NEXT` is reached, 1 is added to `x` (the default `STEP`) and, because `x` is less than 10, execution jumps to the `FOR` statement. On the second loop, `x` is 2 and so on. When `x` reaches 10, `NEXT` adds on 1 and the exit condition is met. After the loop terminates, the final value of `x` is `11`.

In practice, *`variable`* can start and end at any value and be changed by any value on each loop. For example...

```
10 FOR r = 0 TO (2 * _PI) STEP (_PI / 20)
20   x = SIN(r)
30   y = COS(r)
40 NEXT
```

Computes 40 x and y coordinates describing a circle

(strictly speaking 41 points because 2*pi is included)

The loop exit condition is checked when the program execution reaches the `NEXT` statement. Therefore the instructions in the loop will always be executed at least once regardless of the exit condition e.g. `FOR x = 1 TO 0` will execute once.

If `NEXT` has a *`variable`* specified, the most recent `FOR` statement with that variable be used as the loop.

If `NEXT` has no *`variable`* specified, the most recent `FOR` statement with any variable will be used as the loop.

In the case of nested `FOR` loops, a list of variables can be specified in the `NEXT` statement, the first of which is the inner-most loop, progressing outward.

`variable` can be read and used during the loop for any purpose. If *`variable`* is written to during the loop, the exit condition will be evaluated when the program execution reaches the `NEXT` statement and, if the exit conditions are met, the loop will terminate. This is one way to terminate a `FOR..NEXT` loop early, see also `BREAK`.

Do NOT jump out of a `FOR..NEXT` loop using `GOTO` unless you always jump back in. The interpreter creates special variables to keep track of `FOR..NEXT` loops which are deleted when the loop exits, so jumping out will result in more and more variables being used and eventually the program will crash due to lack of memory. As long as the loop is terminated by executing the `NEXT` statement, or causing a jump to `NEXT` with `BREAK`, all will be ok.

Examples:

```
10 FOR i = 20 TO 0 STEP -2
20   PRINT i
30 NEXT
```

Counts backwards printing all the even numbers from 20 to 0

```
10 FOR i = 1 TO 5
20   FOR j = 1 to i
30     PRINT i
40 NEXT j, i
```

Nested loops. Prints the values ‘1’ once, ‘2’ twice and so on to 5.

GOSUB
RETURN

`GOSUB` branches to a subroutine and `RETURN` exits the subroutine returning to the next statement after the `GOSUB`.

Subroutines are used to avoid re-writing the same block of code multiple times in a program. Instead the code is 'called' upon when required from each place it is needed and the subroutine jumps back to the calling place when done.

Subroutines can be 'nested'. For example, a program can call subroutine 1 and subroutine 1 can call subroutine 2 with another `GOSUB`. The `RETURN` at the end of subroutine 2 will go back to subroutine 1 and the `RETURN` at the end of subroutine 1 will go back to the original caller.

Not legal in immediate mode. See also `ON...GOSUB`.

Syntax:

```
GOSUB line
  .
  .
  .
RETURN
```

`line` is the line number where a subroutine can be found. If *`line`* doesn't exist, the pre-run checks should stop you starting the program.

Example:

```
10   PRINT "Out of the subroutine"
20   GOSUB 1000
30   PRINT "And back out again"
40   END
1000 PRINT "In the subroutine"
1010 RETURN
```

Prints “Out of the subroutine”, “In the subroutine”, “And back out again”. The `END` statement on line 40 stops the program from executing the subroutine code again.

GOTO

Jumps to a given line number.

Not legal in immediate mode. See also `ON...GOTO`.

Syntax:

```
GOTO line
```

`line` is the line number to jump to. If *`line`* doesn't exist, the pre-run checks should stop you starting the program.

Example:

```
10 IF x <> 5 THEN GOTO 40
20 PRINT "x is 5"
30 END
40 PRINT "x is not 5"
```

Prints "x is 5" when `x` is 5, otherwise prints "x is not 5".

HELP

Prints a list of valid commands, functions and platform-specific variables.

Not legal while running.

HEX$

Returns a string representation of a numeric value converted to hexadecimal.

See also `STR$`.

Syntax:

```
HEX$(x)
```

`x` is a numeric value. Is rounded down if not an integer.

Example:

```
10 x = 255
20 PRINT x ; " decimal is " ;
   PRINT HEX$(x) ; " hexadecimal"
```

Prints "255 decimal is FF hexadecimal".

IF
THEN
ELSE

Conditionally executes statements based on the result of a conditional expression.

Syntax:

```
IF expression THEN statements [ELSE
statements]
IF expression GOTO line [ELSE statements]
```

If the result of `expression` is non-zero when evaluated, then the `statements` following `THEN` will be executed or the `GOTO` will be followed to `line`.

Otherwise, if `statements` after `ELSE` are present, they will be executed instead.

`statements` are separated with a colon.

Please check the known issues section for problems surrounding string and numeric expression evaluation order.

Examples:

```
10 a$ = "A"
20 b$ = "B"
30 IF a$ = b$ THEN PRINT "Matching" ELSE
   PRINT "Different"
```

Compares two strings and prints “Matching” if they are the same, or “Different” if they are different.

```
10 IF (x < 5) OR (y < 5) THEN x = x + 1 :
   y = y + 1 ELSE PRINT "Done"
```

More complex expression and multiple statements. If `x` or `y` are less than 5, then add one to each. Otherwise print “Done”.

```
10 x = 10
20 IF x > 5 GOTO 40
30 PRINT “Don’t print this”
40 PRINT “Print this line”
```

Example of `GOTO`. Jumps over line 30 and prints “Print this line”.

INKEY$

Returns one character read from the keyboard.

Unlike `INPUT` which buffers the characters and waits for a return key, `INKEY$` returns immediately whether or not a key has been pressed. This allows the script to continue running.

If no key has been pressed, `INKEY$` returns an empty string. If a number of keys have been pressed, `INKEY$` returns the first character then the next character etc.

Examples:

```
10 REPEAT
20   x$ = INKEY$
30 UNTIL x$ = " "
40 PRINT "You pressed the space bar"
```

Prints “You pressed the space bar” when the user presses the space bar.

INPUT

Takes some user input from the terminal. This pauses the program whilst waiting for input, then resumes when the user hits the return key.

Syntax:

```
INPUT [prompt;]var1[,var2]...
INPUT [prompt,]var1[,var2]...
```

prompt is an optional string that prompts the user for the input that is required. Follow *prompt* with a semicolon for it to be printed with a question mark afterwards, or with a comma to suppress the question mark. *var1, var2...* is a list of variables to save the user data into. *var1* is required.

`INPUT` can accept string inputs, but there must only be one and it must be in the last position in the list of input variables. Multiple inputs must be separated by commas and they are saved to their respective variables. If the user inputs less data than required, `INPUT` will prompt again for the missing data.

Do not use this in a deployed script, as the tag will go indefinitely idle waiting for a user input. This is provided solely for testing.

Examples:

```
10 INPUT "Please input two triangle side
   lengths: ", a, b
20 PRINT "Hypotenuse is: " ; SQR(SQ(a) +
   SQ(b))
```

Calculates and prints the hypotenuse of a right-angled triangle from the two other sides as inputs. Example run below...

```
> RUN
Please input two triangle side lengths: 1,
1
Hypotenuse is: 1.41421
```

```
10 INPUT "What is your weight in kilos and
   name"; w, n$
20 PRINT "Hello " ; n$ ; ", you weigh " ;
   w * 2.2 ; " pounds."
```

Asks for a user's name and weight in kilos then prints the name and weight in pounds. Example run below...

```
> RUN
What is your weight in kilos and name? 70,
Bob
Hello Bob, you weigh 154 pounds.
```

INT

Rounds a given numeric value down to the next integer.

For example, 3.9 becomes 3 and -5.1 becomes -6.

See also `FIX` and `ROUND`.

Examples:

```
10 PRINT INT(10.257)
```

Prints "10".

```
10 PRINT INT(-19.1)
```

Prints "-20".

LEFT$

Returns the leftmost n characters of a string.

See also `RIGHT$` and `MID$`.

Syntax:

```
LEFT$(x$, n)
```

x$ is an input string. *n* is the number of characters to read from the left.

If *n* is greater than or equal to the length of *x$* then the whole string is returned.

Example:

```
10 PRINT LEFT$("Hello world", 5)
```

Prints "Hello".

LEN

Returns the number of characters in a string.

Non printing characters and spaces are counted.

Example:

```
10 PRINT LEN("Hello ")
```

Prints “6”.

LET

Assigns a value to a variable.

Syntax:

```
[LET] variable = expression
```

`variable` can be of numeric type e.g. "A", of string type e.g. "A$" or be an array element e.g. "A(2)". Constants cannot be assigned to.

`expression` can contain variables, constants, functions, arithmetic, logical and relational operators plus brackets and type conversion functions. However the result of the calculation must be of the same type as `variable`. An attempt to assign a string value to a numeric variable or vice-versa will give a "Type Mismatch" error.

`LET` is the only optional keyword in BASIC. The absence of a keyword at the start of the line implies `LET`.

The following two statements have the same effect:

```
LET x = 5

x = 5
```

LIST

Displays the current program.

Not legal while running.

Syntax:

```
LIST [start][-][end]
```

start and *end* are optional values that specify line numbers to print between.

`LIST`	lists all lines of the program
`LIST start`	lists just the one line specified
`LIST start-end`	lists all lines from *start* to *end*
`LIST start-`	lists all lines from *start* onwards
`LIST -end`	lists all lines up to *end*

The specified line numbers don't need to exist in the program, `LIST` will choose the closest start and end lines.

LOAD

Loads an EMBASIC script in the emulation.

Always returns "Error: File not found" on Mataki-Lite tags, as there is no file system. Use the script loader utility to load new scripts on to tags.

Not legal while running. See also `SAVE`.

Syntax:

```
LOAD "filename"
```

On the tag emulation, loads an program with the name *filename.bas*. The file path can be included in the name if it is not in the current directory. Returns "Error: File not found" if the file could not be located.

Do not include the .bas extension for the script in *filename*.

LOG

Returns the natural logarithm of the given numeric value.

Given value must be positive.

Example:

```
10 PRINT LOG(10)
```

Prints “2.30259”.

LOWER$

Returns the given string with all upper-case characters converted to lower-case.

See also `UPPER$`.

Example:

```
10 PRINT LOWER$("HeLlO wOrLd”)
```

Prints “hello world”.

MID$

Returns part of a string starting from a given position and length.

See also `LEFT$` and `RIGHT$`.

Syntax:

```
MID$(x$, n, length)
```

x$ is the string to be operated on. *n* is the position to start the substring. *length* is how many characters to read to make the returned substring.

If *n* is less than 1, then *n* is considered the same as 1. If *n* is greater than the length of *x$* then *n* is considered the same as the position of the last character of *x$*. If *length* is greater than the remaining length of the string, then the returned substring is from *n* to the end of *x$*.

Examples:

```
10 x$ = "Alice, Bob"
20 PRINT MID$(x$, 1, 5)
30 PRINT MID$(x$, 8, 100)
```

Prints "Alice" and "Bob".

NEW

Deletes the program currently stored and all the variables in use. On Mataki-Lite this does not remove the script from the flash memory.

Not legal while running.

Example:

```
> LIST
10 PRINT "A script about to be deleted"
20 x = 1
30 y = 2

> NEW
> 10 PRINT "New script!"
> LIST
10 PRINT "New script!"
```

OLD

Restores a program deleted by `NEW`.

`OLD` is only valid immediately after `NEW`. As soon as any program lines are typed or a program is loaded, `OLD` cannot restore the previous program.

Not legal while running.

Gives error "Can't restore old program" if unable to load the old program.

ON

GOTO
GOSUB

Jump to a different line or subroutine in a list depending on the result of an evaluated expression.

Not legal in immediate mode. See also `GOTO` and `GOSUB`.

Syntax:

```
ON expression GOTO line1[,line2]...
ON expression GOSUB line1[,line2]...
```

Follow the `GOTO`/`GOSUB` to the line at the position evaluated by *`expression`* in the list of lines.

If *`expression`* evaluates to a value not in the list of lines, then the program continues.

Examples:

```
10 x = 1
20 ON x GOTO 40, 60
30 END
40 PRINT "x was 1"
50 END
60 PRINT "x was 2"
```

If `x` is 1 then print “x was 1”, if instead `x` is 2 then print “x was 2”. Any other value of `x` does nothing.

```
10   FOR x = 1 TO 3
20     ON x GOSUB 1000, 1100, 1200
30   NEXT
40   END
1000 PRINT "First message"
1010 RETURN
1100 PRINT "A different message"
1110 RETURN
1200 PRINT "One more for good luck"
1210 RETURN
```

Loops through all values of `x` from 1 to 3 printing different messages.

POS

Returns the current position of the text cursor on the screen where position 1 is the first (left most) column.

Syntax:

```
POS(x)
```

x is a dummy parameter and can be any value.

Example:

```
10 PRINT "The position of this space ->";
20 x = POS(0)
30 PRINT " <- is " ; x
```

Prints "The position of this space -> <- is 30"

PRINT

Outputs characters to the PuTTY terminal.

Syntax:

```
PRINT [expression1][;/,]
      [expression2][;/,]...
```

All expressions are evaluated before being printed.

Expressions can be separated with:

;	Next expression is printed immediately after the last
Space(s)	Next expression is printed immediately after the last
,	Next expression is printed in next tab stop (next multiple of 8 chars from the start, which is useful for lining things up in columns)

If no expressions are given, then a blank line is printed.

Unless the statement ends with a semicolon or comma, a carriage return/line feed is automatically added.

A question mark can be used instead of the `PRINT` command in immediate mode. This is converted to a `PRINT` when stored in the script.

Example:

```
10 h$ = "Hello"
20 w$ = "world"
30 PRINT "Hello world"
40 PRINT "Hello " "world"
50 PRINT "Hello " ; "world"
60 PRINT h$ ; " " ; w$
```

Prints "Hello world" a number of times, even though each has a slightly different approach.

```
10 a$ = "Alice"
20 b$ = "Bob"
30 x$ = "Apple pie"
40 y$ = "Biscuits"
50 PRINT "Name" , "Favourite food"
60 PRINT "-------------------------"
70 PRINT a$ , x$
80 PRINT b$ , y$
```

Prints a table of people and their favourite foods.

RANDOMIZE

Re-seeds the random number generator used by `RND`.

`RANDOMIZE` controls the initial starting point for random numbers generated by `RND`. However different tasks require different types of random number.

Sometimes you want a pseudo-random sequence that is the same every time. For example to set a board with random pieces but have two players use the same board, the seed value used by `RANDOMIZE` is effectively the board number. With the same board number you always get the same pattern of pieces.

Other times you want a truly random pattern of numbers which is different each time. For this, use `CLOCK` as the seed.

Example:

```
10 RANDOMIZE(123)
20 PRINT RND
```

Prints "0.339305"

READ

Used to read information from `DATA` statements. `DATA` statements can be anywhere in the program

Syntax:

```
READ var1[,var2]...
```

The list of variables corresponds to a subset of the information stored by `DATA`. The type of data must match the variable type or a "Type Mismatch" error will occur.

Multiple uses of `READ` will run through the available `DATA`, incremented by the number of variables read. See the third example to see how this works.

See `RESTORE` on how to move back to the start of the current `DATA` or change where `DATA` can be found.

If the trying to read past the end of `DATA` then the error "Out of DATA" is given.

Example:

```
10 DATA "Apple", "Banana", "Cherry"
20 READ A$, B$, C$
30 PRINT A$, B$, C$
```

Prints the types of fruit stored by `DATA`

```
10 READ N$, H
20 DATA "Alice", 158
30 PRINT N$ ; " - " ; H ; "cm"
```

An example where `READ` has been used before `DATA` and multiple data types have been used. Prints "Alice - 158cm"

```
10 DATA "Alice", 158, "Bob", 176,
   "Carol", 164
20 FOR i = 1 TO 3
30   READ N$, H
40   PRINT N$ ; " - " ; H ; "cm"
50 NEXT
```

An extension to the last example where `DATA` is traversed with multiple calls to `READ` and prints "Alice - 158cm", "Bob - 176cm", and "Carol - 166cm".

REM

Allows explanatory remarks to be made in the program.

Syntax:

```
REM comment
```

The text in *`comment`* and any statements after *`comment`* are ignored/skipped when running. This is so that `REM` can be used to temporarily disable a line of code without losing the code.

`REM` may also be used at the end of a line. Note that it cannot be used at the end of a `DATA` line, as it would be considered valid data.

Example:

```
10 REM This is a regular comment
20 REM PRINT "Don’t print this"
30 PRINT "Print this" REM Comment after a
   line
```

Prints “Print this”, ignoring valid code after `REM`.

REPEAT
UNTIL

Executes a series of lines until a final condition is met.

Syntax:

```
REPEAT
   .
   .
   .
UNTIL expression
```

The lines between `REPEAT` and `UNTIL` will be executed at least once. *`expression`* is evaluated when `UNTIL` is reached. If *`expression`* evaluates to zero, then the current program position moves back to the corresponding `REPEAT`. Otherwise, the program continues.

Take care when writing `REPEAT/UNTIL` loops, as it's easily possible to get stuck inside them forever waiting for conditions that never occur. When waiting for real-world events like a radio message, which may or may not happen, it is imperative there is always a timeout condition in addition to any other conditions. See the second example below.

The `BREAK` command can be used to exit a loop early.

Please check the known issues section for problems surrounding string and numeric expression evaluation order.

Example:

```
10 REPEAT
20   x$ = INKEY$
30 UNTIL x$ <> ""
40 PRINT "You pressed " ; x$
```

Waits for the user to press a key then prints the key pressed.

```
10 timeout = CLOCK + 3
20 REPEAT
30   x$ = INKEY$
40   IF x$ <> "" THEN BREAK
50 UNTIL CLOCK > timeout
60 IF x$ = "" THEN PRINT "Too slow!" ELSE
   PRINT "Congratulations!"
```

Press any key before 3 seconds elapse to get a congratulated. Demonstrates a timeout.

RESTORE

Moves the point `READ` gets `DATA` from.

Syntax:

```
RESTORE [line]
```

When *`line`* is omitted, the current `DATA` read point is moved back to the first `DATA` statement in the program.

If *`line`* is specified, then the current `DATA` read point is moved to the next `DATA` statement on or after *`line`*.

If the *`line`* cannot be found, then the error “Line not found” is given.

Examples:

```
10 DATA One, Two, Three
20 FOR i = 1 TO 3
30   READ x$
40   PRINT x$
50 NEXT
60 RESTORE
70 GOTO 20
```

Prints “One”, “Two”, and “Three” repeatedly forever. Line 60 causes the `READ` command to start back at “One”.

```
10 DATA One, Two, Three
20 DATA Four, Five, Six
30 READ x$, y$, z$
40 PRINT x$ ; y$ ; z$
50 RESTORE 20
60 READ x$, y$, z$
70 PRINT x$ ; y$ ; z$
```

Prints “One”, “Two”, “Three”, “Four”, “Five”, and “Six”. This demonstrates multiple lines of `DATA`, switched between with `RESTORE`.

RIGHT$

Returns the rightmost n characters of a string.

See also `LEFT$` and `MID$`.

Syntax:

```
RIGHT$(x$, n)
```

x$ is an input string. *n* is the number of characters to read from the right.

If *n* is greater than or equal to the length of *x$* then the whole string is returned.

Example:

```
10 PRINT RIGHT$("Hello world", 5)
```

Prints “world”.

RND

Returns a random number between 0 and 1.

Syntax:

```
RND[(n)]
```

If *n* is omitted, then the value returned will be the next random number. If *n* is 0, then the last random number generated is returned.

Seed the random number generation with `RANDOMIZE`.

Examples:

```
10 RANDOMIZE(123)
20 PRINT RND
30 PRINT RND(0)
```

Prints “0.339305” twice.

```
10 x = 100
20 PRINT INT(RND * (x + 1))
```

Prints a random integer between 0 and `x` (inclusive).

ROUND

Rounds the given numeric value to the nearest integer.

0.5 (exactly) rounds up to 1 and -0.5 rounds down to -1.

See also `FIX` and `INT`.

Example:

```
10 PRINT ROUND(4.5)
20 PRINT ROUND(12.246)
30 PRINT ROUND(-199.85)
```

Prints "5", "12" and "-200".

RUN

Starts execution of program from the first line. Also performs a pre-run check to evaluate any problems before starting.

Not legal while running.

Example:

```
10 PRINT "Hello"

> RUN
Hello
```

SAVE

Saves the current program so it can be recalled later.

Not legal while running. See also `LOAD`.

Syntax:

```
SAVE "filename"
```

`filename` is the name the file should be given and, in the emulation, can include a directory path. Do not include the ".bas" extension.

On Mataki-Lite, this saves the current program over the one stored in flash. The "uploaded" field on start up is set to "Local SAVE". Note that auto-run cannot be set when saving locally on a device.

Example:

Imagine a Mataki-Lite with the current script name old_script.bas:

```
10 PRINT "I am a script"
20 x = 10
```

Then the user makes edits to the script such that it looks like so:

```
10 PRINT "I am an edited script"
20 x = 50
```

Then executes the following command on the command line:

```
> SAVE "new_script"
```

Then on the next device boot, the device would show "new_script.bas" as the current script in the start up message and using `LIST` would show the new script:

```
> LIST
10 PRINT "I am an edited script"
20 x = 50
```

SGN

Returns the sign of a given numeric value.

`SGN(x)` for different values of `x`:

`x` > 0	Returns 1
`x` = 0	Returns 0
`x` < 0	Returns -1

Example:

```
10 PRINT SGN(5) ; ", " ;
20 PRINT SGN(0) ; ", " ;
30 PRINT SGN(-81.24)
```

Prints "1, 0, -1"

SIN

Returns the trigonometric sine of a numeric value in radians.

See also `COS`, `TAN` and `ATN`.

Syntax:

```
SIN(x)
```

`x` is a value in radians.

To convert the return value to degrees, multiply by `_RADTODEG`.

Example:

```
10 PRINT SIN(0)
20 PRINT SIN(_PI / 2)
30 PRINT SIN(_PI)
40 PRINT SIN(3 * _PI / 2)
```

Prints "0", "1", "0", and "-1".

SPC

Used to insert spaces when using a `PRINT` command.

If more spaces are inserted than the terminal has width to display them, then they are continued on the next line.

See also `TAB`.

Example:

```
10 PRINT "A" ; SPC(10) ;
20 PRINT "B" ; SPC(10) ;
30 PRINT "C"
```

Prints “A   B   C”.

SQ

Returns the square of a numeric value.

See also `SQR`.

Example:

```
10 PRINT SQ(5)
20 PRINT SQ(-8.433)
30 PRINT SQ(SQR(2))
```

Prints “25”, “71.1155”, and “2”.

SQR

Returns the square root of a numeric value.

If the value given is less than 0, then the error "Square root of negative number" is given.

See also `SQ`.

Example:

```
10 PRINT SQR(25)
20 PRINT SQR(2)
30 PRINT SQR(0)
```

Prints "5", "1.41421", and 0.

STOP

Stops program execution and returns to immediate mode, where it can be continued with `CONT`.

Not legal in immediate mode. See also `END`.

Gives the message "STOP at line n" when encountered, where n is the line where the `STOP` command was found. In contrast, `END` simply stops silently.

Do not use this in any deployed script, as going back to immediate mode would make the tag go indefinitely idle.

Example:

This program prints "Hello" then stops.

```
10 PRINT "Hello"
20 STOP
30 PRINT "Goodbye"
```

When run, it gives the following output...

```
> RUN
Hello
STOP at line 20
```

STR$

Returns a string representation of a given numeric value in decimal.

See also `HEX$`.

Example:

```
10 x = 5
20 y = -12
30 z = 45.12
40 LEN(STR$(x)) ;
50 LEN(STR$(y)) ;
60 LEN(STR$(z))
```

Gives the number of characters in some different numbers. Prints "1, 3, 5".

SWAP

Swaps the values of two variables of the same type.

Syntax:

```
SWAP var1, var2
```

If the types of *var1* and *var2* are not the same then the error "Type Mismatch" is given.

Example:

```
10   a$ = "Alice"
20   b$ = "Bob"
30   x = 28
40   y = 31
50   GOSUB 1000
60   SWAP x, y
70   GOSUB 1000
80   SWAP a$, b$
90   GOSUB 1000
100  END
1000 PRINT a$ ; " is " ; x ; " and " ;
1010 PRINT b$ ; " is " ; y
1020 RETURN
```

Prints "Alice is 28 and Bob is 31", "Alice is 31 and Bob is 28", and "Bob is 31 and Alice is 28".

SYSTEM

Exits the EMBASIC interpreter on the emulator. Has no effect on Mataki-Lite.

Not legal while running.

TAB

Used to move the text cursor to a specific position when using a `PRINT` command.

See also `SPC`.

Syntax:

```
TAB(x)
```

If `x` is less than 1, then it is set to position 1. If `x` is greater than 80, then it is set to position 80. If `x` is further back than the current text cursor position, then it remains where it currently is.

Examples:

```
10 PRINT "Hello" ; TAB(10) ;
20 "world" ; TAB(20) ; "!"
```

Prints "Hello   world   !" (characters starting at cursor positions 0, 10, and 20).

```
10 PRINT "Hello" ; TAB(1) ; "world"
```

Prints "Helloworld", because the `TAB(1)` tries to move the cursor to a position it has already passed.

TAN

Returns the trigonometric tangent of a numeric value in radians.

See also `ATN`, `SIN` and `COS`.

Syntax:

```
TAN(x)
```

`x` is a value in radians.

To convert the return value to degrees, multiply by `_RADTODEG`.

Example:

```
10 PRINT TAN(0)
20 PRINT TAN(_PI / 4)
```

Prints "0" and "1".

TIME$

Returns a string representation of the time in the format "HH:MM:SS". This is a read-only value.

All time/date values are in UTC because the time is obtained from the GPS satellites.

See also `DATE$`.

Example:

```
10 PRINT TIME$
```

Prints the current time.

TIMER

An integer containing the number of seconds since midnight. This is a read-only value.

All time/date values are in UTC because the time is obtained from the GPS satellites.

`TIMER` is intended to be used in applications where the same thing happens every day at a prescribed time e.g. only taking GPS fixes at times when an animal is expected to be awake.

`TIMER` should not be used to measure the duration of events e.g. for timeout purposes as the value resets to zero at midnight. For this type of application, use `CLOCK`.

Note that, in common with all the other timer functions, if the GPS time has not been obtained e.g. on first switch on, then the time will be wrong and scripts should be written to expect this and anticipate a sudden time jump when the time is obtained.

Example:

```
10 IF TIMER < 79200 PRINT "I want to go to
   sleep" : END
20 PRINT "I'm sleeping..."
```

If it's before 10PM, print "I want to go to sleep". If it's after 10PM print "I'm sleeping...".

TRON
`TROFF`

Enables or disables tracing the line numbers of the commands being executed.

This is a useful debugging tool to find out where the application is going as it is running.

The use of `TRON` anywhere in a program will cause it to be enabled for the runtime of the whole program.

`TROFF` disables line number tracing.

Example:

```
10 PRINT "The first print"
20 x = 1 + 1
30 PRINT "And the second"
40 FOR i = 1 TO x
50   y = 5
60 NEXT

> TRON
> RUN
[10]The first print
[20][30]And the second
[40][50][60][50][60]
```

UPPER$

Returns the given string with all lower-case characters converted to upper-case.

See also `LOWER$`.

Example:

```
10 PRINT UPPER$("HeLlO wOrLd”)
```

Prints “HELLO WORLD”.

VAL

Returns the numerical value of a given string.

If the first non-whitespace character of the input string is not a valid number (-, ., 0-9) then the return value will be undefined.

If any non-numeric characters are found after the start of the string, then the value up to that point is returned.

Examples:

```
10 PRINT VAL("50")
20 PRINT VAL("10.02")
30 PRINT VAL("  -8.7")
40 PRINT VAL(".1")
50 PRINT VAL("123abc")
```

Prints "50", "10.02", "-8.7", "0.1", and "123".

VER

Prints the platform version.

Example:

```
10 VER
```

Prints "V1.2.3".

6.2. Operators

Numeric	
+	Adds two numeric values.
-	Subtracts one numeric value from another.
*****	Multiplies two numeric values.
/	Divides one numeric value by another.
^	Raise a numeric value to the power of another.
MOD	Performs modulo operation on two numeric values. They are truncated to integers before the operation. While negative values are allowed, unintended results may occur if they are used.
()	Brackets may be used to change the order of evaluation

String	
+	Concatenates two string values.

Relational	
=	Compares two values for equality. Returns -1 (true) if the two values are the same or 0 (false) if the two values are different.
<>	Compares two values for inequality. Returns -1 (true) if the two values are different or 0 (false) if the two values are the same.
>	Compares two values in size. Returns -1 (true) if value 1 is greater than value 2 or 0 (false) if value 1 is less than or equal to value 2.
>=	Compares two values in size. Returns -1 (true) if value 1 is greater than or equal to value 2 or 0 (false) if value 1 is less than value 2.
<	Compares two values in size. Returns -1 (true) if value 1 is less than value 2 or 0 (false) if value 1 is greater than or equal to value 2.
<=	Compares two values in size. Returns -1 (true) if value 1 is less than or equal to value 2 or 0 (false) if value 1 is greater than value 2.

Relational operators can be used to compare numeric values or strings. When comparing strings, the '<', '<=', '>' and '>=' operators reflect the alphabetical order in ASCII. Due to issues with strings (see section 8) the string relational operators work in `IF` statements but don't return true and false like they should, so combining numeric and string comparisons with logical operators may not work as expected.

Relational operators return a value of 0 if the comparison is false and -1 if the comparison is true e.g. (1 = 2) has a value of 0 (false, one is not equal to two), whereas (1 < 2) has a value of -1 (true, one is less than two). In binary 0 is all bits '0' and -1 is all bits '1'.

Using the bitwise operators below on results of comparisons using relational operators results in the logical operator behaviour, giving rise to naturally expected behaviour for expressions such as...

```
        IF (X < 10) AND (Y > 3) THEN...
```

If `(X < 10)` is true, it has a value of -1. If `(Y > 3)` is true, it also has a value of -1. The bitwise AND of all binary '1's is all '1's i.e. -1. So the whole expression is true. If one of the comparisons is false, the bitwise AND will result in zero (false).

So the logical operator behaviour is ensured by the true and false values returned by relational operators. However, care must be taken not to apply bitwise operators to values which are not true/false. For example NOT 1 has a value of -2, because 1 is neither true nor false.

In the table below, the bitwise behaviour of each operator is described and the more natural logical behaviour (as will result from comparisons using relational operators).

Bitwise / Logical	
NOT	**Bitwise**: Unary operator returning the one's complement of the given value (every bit inverted). **Logical**: A true value becomes false and a false value becomes true.
AND	**Bitwise**: Combines the two values with a bitwise AND operation. **Logical**: True if both values are true, otherwise false.
OR	**Bitwise**: Combines the two values with a bitwise OR operation. **Logical**: True if either or both values are true, otherwise false.
XOR	**Bitwise**: Combines the two values with a bitwise XOR operation. **Logical**: True if only one of the values is true (not both), otherwise false. XOR can also be described as true if the values are different, false if they are the same.

6.3. Operator Precedence

When evaluating expressions, EMBASIC applies the operators in the order shown in the table below. This is in line with standard mathematical practice and is common to most programming languages. The order can always be changed using brackets. However, the reason for operator precedence is to allow the programmer to write an expression in a natural way and have it operate how most people would expect *without needing brackets*.

If EMBASIC evaluated expressions from left to right with no operator precedence, the following expression...

```
IF X < 10 AND Y > 3 THEN...
```

would be interpreted as...

```
IF ((X < 10) AND Y) > 3 THEN...
```

but with operator precedence, because `AND` has a lower precedence than the `>` operator, the expression is interpreted as...

```
IF ((X < 10) AND (Y > 3)) THEN...
```

...which is the naturally expected behaviour.

Experienced programmers use brackets when there is a chance another programmer might misinterpret his/her intention, even though strictly speaking, the operator precedence is known and can be looked up if required.

This example shows how the addition of brackets would make things clearer. The statement prints 25, calculated as (3 * 7) + (2 squared)...

```
PRINT 3 * 7 + 2 ^ 2
```

Highest Precedence	
()	Brackets (not strictly operators)
^	Raise to a power
*** /**	Multiply and divide
MOD	Modulo operation
+ -	Add and subtract
= <> > < >= <=	All the relational operators
NOT	NOT operator
AND	AND operator
OR	OR operator
XOR	XOR operator
Lowest Precedence	

7. Not Supported

This is a list of features that are supported in some versions of BASIC, but aren't currently supported in EMBASIC:

- String arrays (take too much memory)
- Multi-dimensional arrays (can be emulated, generally take too much memory)
- Reading/writing files in a script (no file system on a tag)
- Integer variables (for simplicity, all variables are floating point)
    - e.g. `L%`
- Nested conditional statements
    - e.g. `IF A>B THEN IF C>D THEN E=1 ELSE E=2 ELSE E=0`
- `TAB()` and `SPC()` don't suppress carriage return as the specification requires
- User defined functions (`DEF FN`)
- `WHILE` and `WEND`
- `PRINT USING`

8. Known Issues

This is a list of known issues that are intended to be fixed:

- `IF` and `UNTIL` statements do not support mixed type expressions, e.g.

```
IF (A$ = "Y") AND (B < 3) THEN...
```