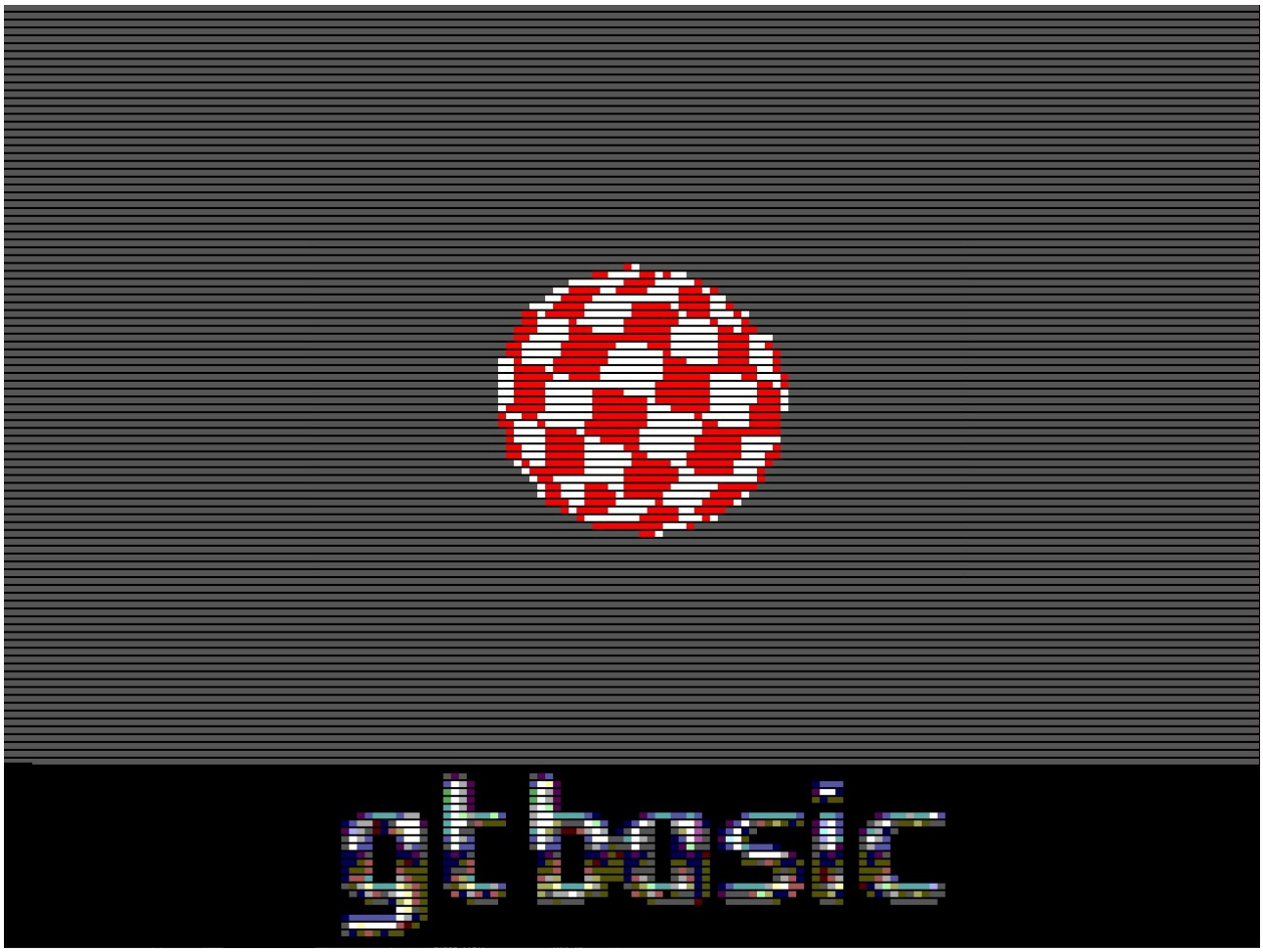
gtBASIC

v1.1.0R, (Release)

Jan-2020, Feb-2021

Ari Tsironis, (at67)



Contents

Statements

ASM

ASM : turns on vCPU assembly code parsing, (which can be disabled with **ENDASM),** during ASM parsing all gtBASIC variables, constants and labels are available by prepending their names with an underscore. Even though **ASM** is an advanced feature it is easy to use and allows simple and flexible integration of vCPU assembly code within gtBASIC code.

```
x = 0

ASM
    LDW    _x
    ADDI   1
    STW    _x
ENDASM
```

AT

AT x or AT x, y : sets the cursor position for the next **PRINT** command and the next **LINE** x, y command. If the ***y*** component is missing then the AT command will use the previously stored ***y*** value, (which defaults to zero).

```
AT 2,0
PRINT “Hello”

AT 38
PRINT “World!”

AT 0,8
LINE 74,0
```

BCDADD

BCDADD src, dst, len : adds ***src*** BCD string to ***dst*** BCD string and stores the result back into ***dst*** BCD string. It expects ***len*** to match both BCD string lengths, ***src*** and ***dst*** are expected to be pointers, (memory addresses), so use @. **Note** a BCD string is not the same as a normal string, see ***Data Types - BCD String***.

```
const SCORE_LEN = 6

DIM scoreBCD%(SCORE_LEN - 1) = 0
DIM pointsBCD%(SCORE_LEN - 1) = 0

‘ BCDADD requires BCD addresses to point to msd, (most significant
‘ digit)
BCDADD @pointsBCD, @scoreBCD, SCORE_LEN
```

BCDCPY

BCDCPY src, dst, len : copies ***src*** BCD string to ***dst*** BCD string. It expects ***len*** to match both BCD string lengths, ***src*** and ***dst*** are expected to be pointers, (memory addresses), so use @. **Note** a BCD string is not the same as a normal string, see ***Data Types - BCD String***.

```
const SCORE_LEN = 6

DIM scoreBCD%(SCORE_LEN - 1) = 0
DIM pointsBCD%(SCORE_LEN - 1) = 0

‘ BCDCPY requires BCD addresses to point to msd, (most significant
‘ digit)
BCDCPY @pointsBCD, @scoreBCD, SCORE_LEN
```

BCDINT

BCDINT dst, x : initialises ***dst*** BCD string with the int16 value ***x***. Length of ***dst*** BCD string must be greater than or equal to ***5***, ***x*** must be in the range of -32768 to 32767, ***dst*** is expected to be a pointer, (memory address), so use @. **Note** a BCD string is not the same as a normal string, see ***Data Types - BCD String***.

```
const SCORE_LEN = 6

DIM scoreBCD%(SCORE_LEN - 1) = 0
DIM pointsBCD%(SCORE_LEN - 1) = 0

‘ BCDINT requires BCD addresses to point to msd, (most significant
‘ digit)
BCDINT @scoreBCD, 100
BCDINT @pointsBCD, 0
```

BCDSUB

BCDSUB src, dst, len : ***src*** BCD string is subtracted from ***dst*** BCD string and the result stored back into ***dst*** BCD string. It expects ***len*** to match both BCD string lengths, ***src*** and ***dst*** are expected to be pointers, (memory addresses), so use @. **Note** a BCD string is not the same as a normal string, see ***Data Types - BCD String***.

```
const SCORE_LEN = 6

DIM scoreBCD%(SCORE_LEN - 1) = 0
DIM pointsBCD%(SCORE_LEN - 1) = 0

‘ BCDSUB requires BCD addresses to point to msd, (most significant
‘ digit)
BCDSUB @pointsBCD, @scoreBCD, SCORE_LEN
```

CALL

CALL <proc name>, param0, param1, … paramN : Call's a procedure with optional parameters, the parameters must be int16 variables and the total amount of them cannot be greater than ***8***. See **LOCAL**, **PROC** and **ENDPROC** for more information.

```
CALL eraseInvaderEdge, x, y
```

CIRCLE

CIRCLE x, y, r : draws the outline of a circle using the current foreground colour, (see **SET FGCOLOUR**), at position ***x, y*** with radius ***r.***

```
‘ white
SET FGCOLOUR,&h3F

‘ circle at the center of the screen with radius 40
CIRCLE 160,120,40
```

CIRCLEF

CIRCLEF x, y, r : draws a filled circle using the current foreground colour, (see **SET FGCOLOUR**), at position ***x, y*** with radius ***r.***

```
‘ blue
SET FGCOLOUR,&h30

‘ filled circle at the center of the screen with radius 40
CIRCLEF 160,120,40
```

CLS

CLS, CLS INIT, CLS <address>, <optional width>, <optional height> :

CLS clears the entire screen.

CLS INIT initialises the screen, (VTop and Video Indirection Table), but does not clear the screen.

CLS <address>, <optional width>, <optional height> can be used to clear any rectangular region of RAM, useful for double buffering or clearing image buffers.

```
CLS
CLS INIT
CLS &h0800,160,120
```

CONST

CONST x | s$ = <number> | <“string”> : defines the entity following it as a literal constant either taking up no memory space, (int vars), or instanced memory space, (strings). Instanced memory space for strings means that all references to that string will reference the exact same string in memory. If you require unique modifiable copies of strings, do not use **CONST** strings.

```
CONST fontstart = 0
CONST boingstart = fontstart + 27
LOAD SPRITE, ../../images/gbas/Boing/Boing0.tga, boingStart + 0
LOAD SPRITE, ../../images/gbas/Boing/Boing1.tga, boingStart + 1

‘ CONST allows you to easily reference an instanced copy of a literal
‘ string anywhere in your code
CONST name$ = "insert name here"

PRINT name$
IF s$ = name$ THEN GOSUB doSomething
```

DATA

DATA <data0>, <data1>, … , <dataN> : is used to define data that can be read into variables defined as part of the **READ** statement. These statements are provided as support for backwards compatibility with older versions of BASIC. Data stored in a **DATA** statement must be of int16 or string types, variables defined or used in a **READ** statement must be int16, string or arrays of int16’s and string’s. **DATA** and **READ** are a poor way of initialising variables and data structures in terms of code space, code cycles and memory usage. The data is effectively stored twice within RAM, once in the **DATA** statements and then once within the **READ** variables. It is much more efficient to statically declare data, which requires no code space, code cycles or a second copy of the data in RAM; see ***Tips, Tricks and Hints – Static Initialisation***.

```
DATA 1, "one", 2, "two", 3, "three", 4, "four", 5, "five", 6, "test", 7,
"yes", 8, "no"

FOR i=1 TO 4
   READ a,a$
   PRINT i;" ";HEX$(a,4);" : ";a$
NEXT I
```

DEC

DEC <var> : decrements either the low byte of an int16 variable, the high byte of an int16 variable or the entire 16bits of that variable if you are using **ROMvX0** or greater, (**Note, ROMv5a** and **DEVROM** do not count as greater than **ROMvX0**). If you are using **ROMv1** to **ROMv4** or **ROMv5a** and **DEVROM** then **DEC** can only decrement the entire 16bits of an int16 variable and you will receive a version error when trying to compile code that does otherwise.

```
‘ ok for all ROM versions
DEC a

‘ the following only works on ROMvX0
DEC a.lo
DEC a.hi
```

DEF

DEF BYTE(); DEF WORD(); DEF <var0>, <var1>, … , <varN>; DEF FN <f> : is a complex statement that offers multiple ways to define static memory initialisations for look up tables, arrays and variable lists. When combined with the statement **FN** it also allows user made functions to be created and executed, see ***Tips, Tricks and Hints – DEF magic*** for more information and examples.

```
‘ static initialisation
DEF BYTE(&h0600) = 1,2,3,4,5
DEF WORD(&h0600) = -1,2000,3000,4000,5000

‘ complex looped intialisation for LUT’s, etc
DEF BYTE(&h0600, y, 0.0, 1.0, 32) = exp(-5.0*pow(y, 3.0))*32.0
DEF WORD(&h0600, y, 0.0, 1.0, 32) = exp(-3.0*y)*6000.0 + 14000.0

‘ user defined function
j = 1
DEF FN func(x, y, z) = x + y + z + j
PRINT func(8, 8, 8) + func(1, 1, 1) + func(3, 3, 3)
```

DIM

DIM : dimensions, (creates), arrays of byte, int16 and strings. The minimum number of dimensions for all types is ***1,*** the maximum number of dimensions for arrays of byte’s and int16’s is ***3*** and the maximum number of dimensions for arrays of string’s is ***1***. Byte arrays are allocated by postfixing **%** to the variable’s name, this is the only time the **%** postfix is used. String arrays are allocated by postfixing **$** to the variable’s name. The amount of entries defined per dimensions is always ***n + 1***, the entries are accessed per dimension via indices ***0*** to ***n***.

```
‘ byte array representing the topology of a string, creates 5 bytes
DIM b%(4) = 3, 'C', 'A', 'T', 0
PRINT b(1);” “;b(2);” “;b(3)
PRINT STRING$(@b)

‘ word array, creates 3 int16’s
DIM w(2) = rnd(0), sin(-90)*10000, -1

‘ string array, creates 3 strings
DIM s$(2) = “One”, “Two”, “Three”

‘ multi-dimensional array, creates 2*2*2 = 8 int16’s
DIM m(1,1,1) = 0, 1, 2, 3, 4, 5, 6, 7
PRINT m(0,0,0);” “;m(1,1,1)
```

DOKE

DOKE a, w : stores ***w,*** (a 16bit value &h0000 to &hFFFF), into the memory location at address ***a***.

```
‘ sets the 2 top left hand pixels to white
const a = &h08a0
DOKE a,&h3F3F
```

ELSE

ELSE : provides an alternate path of execution for a condition statement, used in **IF ENDIF** blocks.

```
k = GET(“SERIAL_RAW”)
IF (k = ‘Y’) OR (k = ‘y’)
    PRINT “Yes”
ELSE
    PRINT “No”
ENDIF
```

ELSEIF

ELSEIF <condition> : provides an alternate path of execution for a condition statement, but also provides it’s own condition statement; used in **IF ENDIF** blocks.

```
k = GET(“SERIAL_RAW”)
IF k = ‘1’
    PRINT “ONE”
ELSEIF k = ‘2’
    PRINT “TWO”
ELSEIF k = ‘3’
    PRINT “THREE”
ELSE
    PRINT “NONE”
ENDIF
```

END

END : terminates or ends or halts the current program, the Gigatron must be reset after executing an **END** statement.

```
loop:
    PRINT “FREEEEEEEEEEDOM!”
    k = GET(“SERIAL_RAW”)
    IF k = ‘q’ THEN END
GOTO loop
```

ENDASM

ENDASM : turns off vCPU assembly code parsing, during ASM parsing all gtBASIC variables, constants and labels are available by prepending their names with an underscore, used in conjunction with **ASM**.

```
x = 0

ASM
    LDW    _x
    ADDI   1
    STW    _x
ENDASM
```

ENDIF

ENDIF : terminates an **IF ENDIF** block, see **IF**.

```
k = GET(“SERIAL_RAW”)
IF (k = ‘Y’) OR (k = ‘y’)
    PRINT “Yes”
ELSE
    PRINT “No”
ENDIF
```

ENDPROC

ENDPROC : terminates a **PROC ENDPROC** block, see **PROC, LOCAL** and **CALL**.

```
PROC drawPlayer
    IF pflip = 1
        sprite FlipX,  Player + 1, px, py
    ELSE
        sprite NoFlip, Player + 0, px, py
    ENDIF
ENDPROC
```

EXEC

EXEC a, <optional r> : loads vCPU code stored at ROM address ***a*** and executes it at optional RAM address ***r***, if ***r*** is missing, then execution begins in RAM at &h0200. Uses the internal ROM Sys function, ‘*SYS_Exec_88’* to perform it’s magic.

```
‘ this code will not work without a ROM address that contains valid vCPU
‘ code, &h1800 will crash the Gigatron
a = &h1800
EXEC a, &h0200
```

FOR

FOR <var>=a TO | UPTO | DOWNTO b STEP c : loops from ***a*** to ***b*** stepping by ***c***.

<***var***> will be automatically created if it has not already been defined or created.

TO is equivalent to **UPTO** and is used for loops where ***a*** is smaller than ***b***, you may use the ampersand '***&***' hint to control branch optimisations on the **FOR NEXT** loop.

DOWNTO is used for loops where ***a*** is bigger than ***b***.

STEP and ***c*** are optional and assume a value of ***1*** if they are missing, the **STEP** value ***c*** will almost always be a postive value as the sign of the loop's direction is governed by **TO** and **DOWNTO**. Do not use negative **STEP** values with **TO**, there are corner cases where the for loop will end up in an infinite loop depending on ***a*** and ***b,*** use **DOWNTO** instead.

```
‘ loops 10 times printing digits 0 to 9
FOR i=0 TO 9 : PRINT i : NEXT i

‘ draw a vertical line from bottom to top in the center of the screen
FOR Y=119 DOWNTO 0 : PSET 79,i : NEXT i

‘ prints int16’s from -1000 to 1000 in increments of 100
FOR x=-1000 TO 1000 STEP 100 : PRINT x : NEXT x

‘ prints int16’s from 1000 to -1000 in increments of 10
FOR x=1000 DOWNTO -1000 STEP 10 : PRINT x : NEXT x

‘ FOR NEXT can use variables for it’s parameters
a=50 : b=100 : c=2
FOR i=a TO b STEP c : PRINT i : NEXT i

‘ FOR NEXT with an optimised branch, (Validater will verify the branch
‘ and destination are within the same 256 byte page
FOR i=0 &TO 100 : PRINT i : NEXT i
```

FUNC

FUNC : not yet implemented.

GOSUB

GOSUB <line number> | <label> | <var> | <expression>, <optional default label> : jumps to a subroutine that must end with a **RETURN** statement, the **RETURN** statement causes the execution of code to recommence immediately after the original **GOSUB** statement.

<***line number***> is a literal value between 1 and 32767.

<***label***> is any valid text label as described under ***Flow Control - Labels - Text Labels***.

<***var***> is any valid variable containing a value between 1 and 32767, the line numbers that <***var***> evaluates to must have a trailing colon '***:***', the colon specifies that this numeric label is a **GOSUB** target.

<***expression***> is any valid expression that results in a value between 1 and 32767, the line numbers that <***expression***> evaluates to must have a trailing colon '***:***', the colon specifies that this numeric label is a **GOSUB** target.

<***optional default label***> is an optional label that is jumped to, (and expected to contain a subroutine), that handles values not explicitly labelled.

```
‘ traditional usage, (don’t do this unless you have to)
10 GOSUB 100
20 GOTO 10

100 PRINT “YO!”
110 RETURN


‘ more modern usage
REPEAT
    GOSUB greetings
FOREVER

greetings:
    PRINT “YO!”
RETURN


‘ using GOSUB like a switch statement
REPEAT
    b = GET("SERIAL_RAW")
    GOSUB b, default
FOREVER

255:             RETURN ‘ idle
127: PRINT “A” : RETURN ‘ button A
254: PRINT “R” : RETURN ‘ right
253: PRINT “L” : RETURN ‘ left
251: PRINT “D” : RETURN ‘ down
247: PRINT “U” : RETURN ‘ up

default:
    PRINT “NO”
RETURN
```

GPRINTF

GPRINTF ”<format string>”, <expr0>, <expr1> … , <exprN> : displays logging and debugging information in the emulator’s console window, this statement will only work when executing your gtBASIC code under the gtemuAT67 emulator.

<***”format string”***> works in a very similar way to C’s printf format string, it supports field widths from ***1*** to ***16*** and the following field types.

%c will print a char.

%b will print a byte.

%d will print an int.

%04x will print a zero padded four digit hex number.

%016i will print a zero padded 16 digit binary number.

%04o will print a zero padded 4 digit octal number.

%s will print a string with a maximum length of 94 characters, the first byte of the string is expected to be the length of the string and the last byte is expected to be a terminating zero.

```
‘ int16 variables and expressions
l = 0
GPRINTF "l=%d l+1=%d", l, l+1

‘ string and array variables
l = 0 : q$ = “Success?”
DIM r$(1) = “Yes”, “No”
GPRINTF "q$=%s : r$=%s", i$, r$(l)

‘ address of variables and arrays
GPRINTF "@l=0x%4x : @q$=0x%4x : @r$(l)=0x%4x", @l, @q$, ADDR(r$(l))

‘ byte and binary variables
GPRINTF "l.lo=%b : l.hi=%b : l=%016i", l.lo, l.hi, l
```

HLINE

HLINE x1, y, x2 : draw’s a horizontal line into video or screen RAM without clipping, it can be used to draw lines into offscreen RAM because of the lack of clipping. **HLINE** uses the current foreground colour, see **SET FGCOLOUR**

```
CONST h$ = "HEADING"
CONST xsiz = len(h$)*6 - 2
CONST ysiz = 8
CONST xpos = 80 - xsiz/2
CONST ypos = 60 - ysiz/2

AT xpos, ypos : PRINT h$
HLINE xpos, ypos+ysiz, xpos+xsiz
```

IF

IF <condition> : performs a condition test and takes one of two execution paths if used with **THEN** or two or more execution paths if used in an **IF ENDIF** block. *Note* **ELSE** can only be used within an **IF ENDIF** block, it cannot be used with **IF THEN**.

<**condition**> may be one of '**=**', '**<**', '**>**', '**<=**', '**>=**', '**<>**', <**condition**> can also be turned into a fast boolean test by using parenthesis around a variable which tests a variable for zero.

```
' IF THEN
IF a = 0 THEN PRINT "A is ZERO"

' IF ENDIF block
k = GET("SERIAL_RAW")
IF k = '1'
    PRINT "ONE"
ELSEIF k = '2'
    PRINT "TWO"
ELSEIF k = '3'
    PRINT "THREE"
ELSE
    PRINT "NONE"
ENDIF

' boolean test, the ampersand, '&', hints at the compiler to use a BNE
IF &(a) THEN PRINT "A is ZERO"
```

INC

INC <var> : increments either the low byte of an int16 variable, the high byte of an int16 variable or the entire 16bits of that variable.

```
' ok for all ROM versions
INC a

' ok for all ROM versions
INC a.lo
INC a.hi
```

INIT

INIT VARS, <var> | TIME, MIDI | MIDIV | <user>, NOUPDATE : is a multi-functional command that is able to initialise previously declared variables, (that used **DEF)**, initialise **TIME**, **MIDI** and <**user**> subroutines for time-slicing ROM's and for vertical blank, '*VBlank'*, ROM's.

VARS specified that the int16 variables beginning with <**var**> will be initialised to zero.

TIME specifies that the timer and time subroutines will regularly be called by gtBASIC functions and statements and be approximately 16.7ms accurate. This allows the functions **GET**("**TIMER**") and **TIME$**() to return correct results.

MIDI specifies that the MIDI player will regularly be called by gtBASIC functions and statements, thus playing music asynchronously to your gtBASIC code in the background.

MIDIV is the same as MIDI but specified a MIDI travk with volume or velocity data.

<**user**> is a user subroutine that will be called at the same frequency as the TIME and MIDI subroutines. When using a ROM that contains '*VBlank'* interrupts, the <**user**> subroutine must not use any gtBASIC owned variables, data or runtime. This usually means that the <**user**> subroutine, (for '*VBlank'* ROM's), must be written in vCPU assembly code and use it's own variable space.

NOUPDATE specifies that the time slicing routines will not be called internall by gtBASIC functions and statements, but that you will perform the update manually using the **TICK** statement. This should be used when your code does not use any gtBASIC functions ot statements at regular intervals, (i.e. called at a high enough frequency), in this case place **TICK** in your inner most loop.

```
' initialise all variables, (to zero), starting at @ticks
DEF lives, ticks, flags
INT VARS @ticks

‘ initialise user subroutine ‘scrollSanta’ and MIDI player, scrollSanta
‘ is called first then MIDI player; this is the same syntax for all
‘ ROM’s, whether they are timeslice ROM’s or Vblank ROM’s
INIT scrollSanta, MIDI

‘ initialise TIME, MIDIV, (MIDI with volume/velocity) and user
‘ subroutine ‘userProc’; this is the same syntax for all
‘ ROM’s, whether they are timeslice ROM’s or Vblank ROM’s
INIT TIME, MIDIV, userProc


‘ initialise TIME but disable timeslicing during system calls, you must
‘ do the updates manually using TICK
INIT TIME, NOUPDATE

REPEAT
    TICK TIME
FOREVER
```

INPUT

INPUT <”heading”>, <var0>, <”prompt0”>;fw0;, <varN>, <”promptN”>;fwN; : is a multi-functional statement or command that is create and initialise int16 and string variables, (array variables must have been previously dimensioned using DIM beforehand. Semi-colons ‘**;**’ are used to indicate whether newlines are inserted before and after prompt strings. Input text will scroll when typing hits the edge of the screen, upto the maximum specified by field width or the maximum length of a string, which is currently ***94***.

<”***heading***”> is a heading string used to name or highlight the **INPUT** command.

<***var***> may be an int16, or a string or an array variable and will be initialised with whatever values are processed by the **INPUT** command. Values resulting in errors will return default values, 0 for int16 and empty strings“” for strings, (this is a work in progress, eventually an **ON ERROR** pattern will be ceated to facilitate more flexible user error handling.

<”***prompt***”> is a prompt string attached to the variable preceding it.

<**;*fw*;**> field width specifies how many characters the input will consume before being blocked, (this includes negative signs when using numeric input, the semi-colons are optional newline controllers.

```
‘ creates and initialises an int16 variable after inputing numeric data
INPUT a

‘ creates and initialises a string variable after inputing string data
INPUT s$

‘ initialises an int16 array variable, array must have been previously
‘ dimensioned after using the DIM command
INPUT a(10)

‘ prints a heading string ‘TESTING INPUT’ and creates and initialise an
‘ int16 variable ‘a’, the semi-colons make sure it happens all on the
‘ same line, the field width of 5 specifies no more than 5 characters
INPUT “TESTING INPUT: “, a,”?”;5;
```

Functions

Integer functions

ABS

ABS(x) : returns the absolute value of an int16, result is 0 to 32767, cannot be used in static initialisation.

```
y = ABS(x)
```

ADDR

ADDR(a) : returns the address of any element within an array, result is &h0000 to &hFFFF, can be used in static initialisation.

```
DIM arr0(10) ‘ one dimensional int16 array
a = ADDR(arr0(5))

DIM arr1%(3,4,5) ‘ three dimensional byte array
a = ADDR(arr1(1,1,1))

DIM sarr$(5) ‘ one dimensional string array
a = ADDR(sarr$(3))
```

ASC

ASC(x) : returns the ASCII value of a character, result is 0 to 255, can be used in static initialisation.

```
c = ASC(‘a’)
c = ASC(CHR$(65))
```

BCDCMP

BCDCMP(src, dst, len) : compares two BCD strings and returns ***-1*** for ***src*** < ***dst***, ***0*** for ***src*** = ***dst*** and ***1*** for ***src*** > ***dst***. It expects ***len*** to match both BCD string lengths, ***src*** and ***dst*** are expected to be pointers, (memory addresses), so use @. It cannot be used in static initialisation. **Note** a BCD string is not the same as a normal string, see ***Data Types - BCD String***.

```
const SCORE_LEN = 6

' BCDCMP requires BCD addresses to point to msd, (most significant
‘ digit)
IF BCDCMP(@score+(SCORE_LEN-1), @high+(SCORE_LEN-1), SCORE_LEN) = 1
    BCDCPY @score, @high, SCORE_LEN
ENDIF
```

CLAMP

CLAMP(x, a, b) : returns an int16 value that is the result of clamping of ***x***, between ***a*** and ***b***, result is -32768 to 32767, cannot be used in static initialisation.

```
y = CLAMP(a,1,500)
```

DEEK

DEEK(a) : returns the word value located at RAM address ***a***, result is &h0000 to &hFFFF, cannot be used in static initialisation.

```
w = DEEK(&h08A0)
```

IARR

IARR() : is an internal function that is not normally used by the gtBASIC programmer, it allows the access of byte and int16 arrays, cannot be used in static initialisation.

LUP

LUP(a, o) : returns the byte value located at ROM address ***a*** plus ***o***, result is &h00 to &hFF, cannot be used in static initialisation.

```
b = LUP(&h0900,2)
```

LEN

LEN(a) : returns the allocated memory size of ***a,*** the length returned may be smaller than the actual memory size allocated due to indirection tables and support data, can be used in static initialisation.

```
l = LEN(str$)
l = LEN(strArr$(1))
l = LEN(arr2(1, 0))
```

MAX

MAX(x, y) : returns an int16 value that is the maxium of ***x*** and ***y,*** result is -32768 to 32767, can be used in static initialisation.

```
y = MAX(x,1)
```

MIN

MIN(x, y) : returns an int16 value that is the minimum of ***x*** and ***y,*** result is -32768 to 32767, can be used in static initialisation.

```
y = MIN(x,1)
```

PEEK

PEEK(a) : returns the byte value located at RAM address ***a***, result is &h00 to &hFF, cannot be used in static initialisation.

```
b = PEEK(&h08A0)
```

POINT

POINT(x, y) : returns a byte value representing the pixel at screen coordinates ***x*** and ***y,*** result is &h00 to &hFF, takes into account the vertical component of the video indirection table, but ignores the horizontal component, cannot be used in static initialisation.

```
p = POINT(80,60)
```

RND

RND(r) : returns a random value from ***0*** to ***r - 1,*** if ***r = 0*** then the return value is from &h0000 to &hFFFF which may then be filtered using your own code. ***Note*** filtering using an AND operator will be much faster than using the inbuilt MOD functionality, can be used in static initialisation.

```
‘ random numbers from 0 to 19
r = RND(20)

‘ much faster random numbers from 0 to 31, (only works correctly for
‘ filter/mask values of for ^2 - 1)
r = RND(0) AND 31
```

SARR

SARR() : is an internal function that is not normally used by the gtBASIC programmer, it allows the access of string arrays, cannot be used in static initialisation.

SGN

SGN(x) : returns the sign of ***x*** as ***-1*** if ***x*** is negative, ***0*** if ***x*** is zero and ***1*** if ***x*** is positive, cannot be used in static initialisation.

```
s = SGN(-1000)
```

STRCMP

STRCMP(src$, dst$) : compares two strings and returns ***-1*** for ***src$*** < ***dst$***, ***0*** for ***src$*** = ***dst$*** and ***1*** for ***src$*** > ***dst$,*** cannot be used in static initialisation. **Note** normally there is no need to explicitly use this function as strings may be compared using the normal relational operators, **<**, **>**, **=**, **<>**, **<=**, **>=**.

```
IF STRCMP(str0$, str1$) = 0 THEN PRINT “Yes”

IF str0$ = str1$ THEN PRINT “Yes”
```

URND

URND(x, y, n, s) : returns a sequence of unique random values, between ***x*** and ***y***, ***n*** times, stepping by ***s***; this allows you to generate non repeating random values that can be used to fill memory, can only be used in static initialisation.

```
‘ fill 32 consecutive word memory locations beginning at &h0600
‘ with random numbers from 2 to 157, stepping by 2, (random numbers
‘ generated will be guaranteed to be at least 2 apart)
DEF WORD(&h600, 0, 1, 32) = URND(2, 157, 32, 2)
```

USR

USR(a) : jumps to and executes the code beginning at memory address ***a***, returns whatever the code executed leaves in vAC, cannot be used in static initialisation.

```
const playNote = 129

‘ vCPU assembly code poked into zero page
poke playNote + 0, 17  ' LDWI $9xx
poke playNote + 2, 9
poke playNote + 3, 127 ' LUP 0
poke playNote + 4, 0
poke playNote + 5, 147 ' INC [playNote + 1]
poke playNote + 6, playNote + 1
poke playNote + 7, 255 ' RET

p = 256 + 252

‘ play a note, (there are much easier ways to make sounds, this just
‘ demonstrates one way of using USR(), normally it is a very bad idea
‘ to be poking vCPU assembly code into zero page)
poke p + 0, usr(playNote) ' frequency low
poke p + 1, usr(playNote) ' frequency high
poke 44, 15               ' duration = 0.25 secs
```

VAL

VAL(s$) : returns an int16 numeric value after attempting to convert the string ***s$***, result is -32768 to 32767, but depends upon the contents of the string, (negative signs are interpreted correctly), can be used in static initialisation.

```
v = VAL(“-1234”)
```

String functions

CHR$

CHR$(a) : returns a string of size ***1*** that represents the ASCII value ***a,*** cannot be used in static initialisation.

```
s$ = CHR$(65)
```

HEX$

HEX$(x, n) : returns a string of size ***n*** that represents the hex value of ***x, n*** is must be 1 to 4, cannot be used in static initialisation.

```
s$ = HEX$(&hAA,2)
s$ = HEX$(&hBEEF,4)
```

LEFT$

LEFT$(s$, n) : returns a string of size ***n*** that represents the left substring from ***0*** to ***n-1*** of ***s$***, cannot be used in static initialisation.

```
‘ prints “cat”
l$ = LEFT$(“catdog”,3)
PRINT l$
```

LOWER$

LOWER$(s$) : returns a string that is a lower-case conversion of ***s$***, cannot be used in static initialisation.

```
‘ prints “catdog”
l$ = LOWER$(“CaTdOg”)
PRINT l$
```

MID$

MID$(s$, i, n) : returns a string of size ***n*** that represents the middle substring from ***i*** to ***i+n-1*** of ***s$***, cannot be used in static initialisation.

```
‘ prints “rat”
m$ = MID$(“catratdog”,3,3)
PRINT m$
```

RIGHT$

RIGHT$(s$, n) : returns a string of size ***n*** that represents the right substring from ***strlen(s$)-n*** to ***strlen(s$)-1*** of ***s$***, cannot be used in static initialisation.

```
‘ prints “dog”
r$ = RIGHT$(“catdog”,3)
PRINT r$
```

SPC$

SPC$(n) : returns a string of size ***n*** that contains ***n*** space, maxium n is ***94,*** cannot be used in static initialisation.

```
s$ = SPC$(10)
```

STR$

STR$(x) : returns a string that represents the numerical value of ***x, x*** must be between -32768 and 32767**,** cannot be used in static initialisation.

```
s$ = STR$(-2345)
```

STRCAT$

STRCAT$(src0$, src1$, … , srcN$) : concatenates ***2*** to ***n*** strings together resulting in an output string that is the sum of the inputs, the output string cannot be longer than ***94*** characters and will be cropped if needed; this function is not normally used by the programmer as string concatenation can be performed using the **+** operator, cannot be used in static initialisation.

```
‘ prints “catratdog”
s$ = STRCAT$(“cat”,”rat”,”dog”)
PRINT s$
```

STRING$

STRING$(a) : returns a string that is created from a memory pointer, use this when manually creating strings programatically**,** cannot be used in static initialisation.

```
‘ a needs to point to a valid string topology, <len><string><0>
a = &h0600
s$ = STRING$(a)
```

TIME$

TIME$() : returns a string that represents the current time**,** cannot be used in static initialisation.

```
t$ = TIME$()
```

UPPER$

UPPER$(s$) : returns a string that is an upper-case conversion of ***s$***, cannot be used in static initialisation.

```
‘ prints “CATDOG”
u$ = UPPER$(“CaTdOg”)
PRINT u$
```

GET

- GET is a function that accepts a string literal describing a system variable to get and an optional number of parameters used by that system variable.

```
v = GET(“<system variable name>”, optional param0, … , optional paramN)
```

- System variable’s available for get.

```
bs = GET(“BUTTON_STATE”)
‘ returns the current BUTTON_STATE register, &h00 >= bs <= &hFF

cm = GET(“CHANNEL_MASK”)
‘ returns the current sound channel mask, &h00 >= cm <= &h03

cx = GET(“CURSOR_X”)
‘ returns the current cursor’s x position, 0 >= cx <= 255

cy = GET(“CURSOR_Y”)
‘ returns the current cursor’s y position, 0 >= cy <= 255

cxy = GET(“CURSOR_XY”)
‘ returns the current cursor’s xy position, &h0000 >= cxy <= &hFFFF

bc = GET(“BG_COLOUR”)
‘ returns the current background colour, &h00 >= bc <= &hFF

fc = GET(“FG_COLOUR”)
‘ returns the current foreground colour, &h00 >= fc <= &hFF

fbc = GET(“FGBG_COLOUR”)
‘ returns the current fg and bg colours, &h0000 >= fbc <= &hFFFF

fc = GET(“FRAME_COUNT”)
‘ returns the current FRAME_COUNT register, &h00 >= fc <= &hFF

ls = GET(“LED_STATE”)
‘ returns the current LED state, 0 >= ls <= 255

lt = GET(“LED_TEMPO”)
‘ returns the current LED tempo, 0 >= lt <= 255

ms = GET(“MEM_SIZE”)
‘ returns the current size of memory, &h80 = 32K, &h00 = 64K

ms = GET(“MIDI_STREAM”)
‘ returns the current MIDI stream address, &h0000 >= ms <= &hFFFF

n = GET(“MIDI_NOTE”, m)
‘ returns the midi note corresponding to the index ‘m’, midi indices
‘ start at 12 and end at 106, (this is the entire range of the Gigatron)

n = GET(“MUSIC_NOTE”, i)
‘ returns the note corresponding to the index ‘i’, music indices start
‘ at 0 and end at 94, (this is the entire range of the Gigatron)
```

- System variable's available for get.

```
r0 = GET("RAND0")
' returns the current RAND0 register, &h00 >= r0 <= &hFF

r1 = GET("RAND1")
' returns the current RAND1 register, &h00 >= r1 <= &hFF

r2 = GET("RAND2")
' returns the current RAND2 register, &h00 >= r2 <= &hFF

rn = GET("ROM_NOTES")
' returns the start address of the ROM note table,
' &h0000 >= rn <= &hFFFF

a = 0 : a = GET("ROM_READ_DIR", a) ' only available in >= ROMv5a
' returns the address of the next main menu application, 'a' is the
' address of the current application or 0 for the start of the list

rt3 = GET("ROM_TEXT32")
' returns the address of the ROM 32+ font table, &h0000 >= rt3 <= &hFFFF

rt8 = GET("ROM_TEXT82")
' returns the address of the ROM 82+ font table, &h0000 >= rt8 <= &hFFFF

rt = GET("ROM_TYPE")
' returns the current ROM type, &h00 >= rt <= &hF8

sch0 = GET("SND_CHN0")
' returns the current address of sound channel0,
' &h0000 >= sch0 <= &hFFFF

sch1 = GET("SND_CHN1")
' returns the current address of sound channel1,
' &h0000 >= sch1 <= &hFFFF

sch2 = GET("SND_CHN2")
' returns the current address of sound channel2,
' &h0000 >= sch2 <= &hFFFF

sch3 = GET("SND_CHN3")
' returns the current address of sound channel3,
' &h0000 >= sch3 <= &hFFFF

a = GET("SPRITE_LUT", s) ' only available in >= ROMv3
' returns the address of the sprite corresponding to the sprite id 's'

sr = GET("SERIAL_RAW")
' returns the current SERIAL_RAW register, &h00 >= sr <= &hFF

st = GET("SOUND_TIMER")
' returns the current sound timer, 0 >= st <= 255

te = GET("TIME_EPOCH")
' returns the current time epoch, where time begins

tm = GET("TIME_MODE")
' returns the current time mode, tm = 12 or tm = 24
```

- System variable's available for get.

```
ts = GET("TIME_S")
' returns the current time's seconds value, 0 >= ts <= 59

tm = GET("TIME_M")
' returns the current time's minutes value, 0 >= tm <= 59

th = GET("TIME_H")
' returns the current time's hours value, 0 >= th <= 12/24

t = GET("TIMER")
' returns the current timer's tick value, 0 >= t <= 32767

t = GET("TIMER_PREV")
' returns the internal timer's tick value, 0 >= t <= 32767

vac = GET("VAC")
' returns the current vCPU accumulator, &h0000 >= vac <= &hFFFF

vlr = GET("VLR")
' returns the current vCPU leaf register, &h0000 >= vlr <= &hFFFF

vpc = GET("VPC")
' returns the current vCPU program counter, &h0000 >= vpc <= &hFFFF

vsp = GET("VSP")
' returns the current vCPU stack pointer, &h00 >= vsp <= &hFF

vbp = GET("VBLANK_PROC") ' only available in >= ROMv5a
' returns the current VBlank interrupt address, &h0200 >= vbp <= &hFFFF

vbf = GET("VBLANK_FREQ") ' only available in >= ROMv5a
' returns the current VBlank interrupt frequency, 1 >= vbf <= 255

vram = GET("V_RAM")
' returns the start address of video memory, &h0000 >= vram <= &hFFFF

vtp = GET("V_TOP")
' returns the address of the V_TOP register, &h0000 >= vtp <= &hFFFF

vtb = GET("V_TABLE")
' returns the address of the V_TABLE register, &h0000 >= vtb <= &hFFFF

vy = GET("VIDEO_Y")
' returns the current VIDEO_Y register, &h00 >= vy <= &hFF

xm = GET("XOUT_MASK")
' returns the current XOUT register mask, &h00 >= xm <= &hFF

xr = GET("X_RES")
' returns the Gigatron's current screen width, &h00 >= xr <= &hFF

yr = GET("Y_RES")
' returns the Gigatron's current screen height, &h00 >= yr <= &hFF
```

SET

- SET is a statement that accepts a constant literal describing a system variable to set and an optional number of parameters used by that system variable.

```
SET <system variable name>, option param0, … , optional paramN
```

- System variable’s available for set.

```
SET BUTTON_STATE, bs
‘ sets the button state register to bs, &h00 to &hFF

SET CHANNEL_MASK, cm
‘ sets the sound channel mask to cm, &h00 to &h03

SET CURSOR_X, cx
‘ sets the cursor x position to cx, 0 to 255

SET CURSOR_Y, cy
‘ sets the cursor y position to cy, 0 to 255

SET CURSOR_XY, cxy
‘ sets the cursor xy position to cxy, &h0000 to &hFFFF

SET BG_COLOUR, bc
‘ sets the background colour to bc, &h00 to &hFF

SET FG_COLOUR, fc
‘ sets the foreground colour to fc, &h00 to &hFF

SET FGBG_COLOUR, fbc
‘ sets the foreground and background colour to fbc, &h0000 to &hFFFF

SET FONT_ID, fid
‘ sets the current font id to fid, 0 to 255

SET FRAME_COUNT, frc
‘ sets the background colour to frc, 0 to 255

SET LED_STATE, ls
‘ sets the LED state to ls, 0 to 255

SET LED_TEMPO, lt
‘ sets the LED tempo to lt, 0 to 255

SET MIDI_STREAM, ms
‘ sets the MIDI stream pointer to ms, &h0000 to &hFFFF

SET SOUND_TIMER, st
‘ sets the sound timer to st, 0 to 255

SET TIME_EPOCH, te
‘ sets the time epoch, (in hours), to te, 0 to 12

SET TIME_MODE, tm
‘ sets the time mode to tm, 12 or 24
```

- System variable’s available for set.

```
SET TIME_S, ts
‘ sets the time seconds to ts, 0 to 59

SET TIME_M, tm
‘ sets the time minutes to tm, 0 to 59

SET TIME_H, th
‘ sets the time hours to th, 0 to 12

SET TIMER, tr
‘ sets the system tick timer to tr, 0 to 32767

SET VBLANK_FREQ, vf ‘ only available in >= ROMv5a
‘ sets the VBlank interrupt frequency to vf, 1 to 255

SET VBLANK_PROC, vp ‘ only available in >= ROMv5a
‘ sets the VBlank interrupt address to vp, &h0000 to &hFFFF

SET VIDEO_TOP, vt ‘ only available in >= ROMv5a
‘ sets the starting scanline for video display to vt, &h00 to &hFF

SET XOUT_MASK, xm
‘ sets the xout mask register to xm, &h00 to &hFF
```

Operators

- Operators are symbols or reserved words that allow gtBASIC to perform logical or arithmetic functions, they accept one or more inputs or parameters and always return an output or result.
- Operators can be classified under the following categories, unary logic, unary math, binary logic, binary math and relational.
- Unary logic operators

```
+5    ‘ invokes the POS operator, which does nothing
-5    ‘ invokes the NEG operator, which performs a 2’s complement
NOT 5 ‘ invokes the NOT operator, which performs a 1’s complement
```

- Unary math operators

```
CEIL(x)    ‘ returns smallest integer >= ‘x’, only accepts floats and
           ‘ only usable in static initialisation
FLOOR(x)   ‘ returns largest integer <= ‘x’, only accepts floats and
           ‘ only usable in static initialisation
POWF(x,y)  ‘ returns ‘x’ raised to the power of ‘y’, only usable in
           ‘ static initialisation
SQRT(x)    ‘ returns the non-negative square root of ‘x’, only usable in
           ‘ static initialisation
EXP(x)     ‘ returns the base e exponential of ‘x’, (e^x), only usable
           ‘ in static initialisation
EXP2(x)    ‘ returns the base 2 exponential of ‘x’, (2^x), only usable
           ‘ in static initialisation
LOG(x)     ‘ returns the natural logarithm of ‘x’, only usable in static
           ‘ initialisation
LOG2(x)    ‘ returns the base 2 logarithm of ‘x’, only usable in static
           ‘ initialisation
LOG10(x)   ‘ returns the base 10 logarithm of ‘x’, only usable in static
           ‘ initialisation
SIN(x)     ‘ returns the sine of ‘x’, (x is in degrees), only usable in
           ‘ static initialisation
COS(x)     ‘ returns the cosine of ‘x’, (x is in degrees), only usable
           ‘ in static initialisation
TAN(x)     ‘ returns the tangent of ‘x’, (x is in degrees), only usable
           ‘ in static initialisation
ASIN(x)    ‘ returns the inverse of sine of ‘x’, (x is in degrees), only
           ‘ usable in static initialisation
ACOS(x)    ‘ returns the inverse of cosine of ‘x’, (x is in degrees),
           ‘ only usable in static initialisation
ATAN(x)    ‘ returns the inverse of tangent of ‘x’, (x is in degrees),
           ‘ only usable in static initialisation
ATAN2(y,x) ‘ returns a four quadrant aware version of ATAN, only usable
           ‘ in static initialisation
RAND(x)    ‘ returns a pseudo random number between 0 and x-1, the
           ’ max value of x is 2^32, only usable in static
           ‘ initialisation
REV16(x)   ‘ returns a 16bit result that is the reverse of ‘x’, only
           ‘ usable in static initialisation
REV8(x)    ‘ returns an 8bit result that is the reverse of ‘x’, only
           ‘ usable in static initialisation
REV4(x)    ‘ returns a 4bit result that is the reverse of ‘x’, only
           ‘ usable in static initialisation
```

- Binary logic operators

```
a AND b ‘ performs a logical AND between ‘a’ and ‘b’, works for static
        ‘ initialisation and for run time execution
a OR b  ‘ performs a logical OR between ‘a’ and ‘b’, works for static
        ‘ initialisation and for run time execution
a XOR b ‘ performs a logical XOR between ‘a’ and ‘b’, works for static
        ‘ initialisation and for run time execution
a LSL n ‘ performs a logical shift left of ‘a’ by ‘n’ bits, works for
        ‘ static initialisation, n<=32, and for run time execution, n<=8
a LSR n ‘ performs a logical shift right of ‘a’ by ‘n’ bits, works for
        ‘ static initialisation, n<=32, and for run time execution, n<=8
a ASR n ‘ performs an arithmetic shift right of ‘a’ by ‘n’ bits, use
        ‘ this instead of LSR when you require sign extension, works for
        ‘ static initialisation, n<=32, and for run time execution, n<=8
‘ Note n has to be a literal for all the shift instructions
```

- Binary math operators

```
x + y   ‘ performs an arithmetic ADD between ‘x’ and ‘y’, works for
        ‘ static initialisation and for run time execution
x - y   ‘ performs an arithmetic SUB between ‘x’ and ‘y’, works for
        ‘ static initialisation and for run time execution
x * y   ‘ performs an arithmetic MUL between ‘x’ and ‘y’, works for
        ‘ static initialisation and for run time execution
x / y   ‘ performs an arithmetic DIV between ‘x’ and ‘y’, works for
        ‘ static initialisation and for run time execution
x MOD y ‘ performs an arithmetic MOD between ‘x’ and ‘y’, works for
        ‘ static initialisation and for run time execution
x ** y  ‘ performs an arithmetic x^y between ‘x’ and ‘y’, works for
        ‘ static initialisation and for run time execution
```

- Relational operators

```
x = y  ‘ performs an is equal test between ‘x’ and ‘y’, returns 0
       ‘ on false and 1 on true, works for static initialisation and for
       ‘ run time execution
x <> y ‘ performs an is not equal test between ‘x’ and ‘y’, returns 0
       ‘ on false and 1 on true, works for static initialisation and for
       ‘ run time execution
x <= y ‘ performs an is less than or equal test between ‘x’ and ‘y’,
       ‘ returns 0 on false and 1 on true, works for static
       ‘ initialisation and for run time execution
x >= y ‘ performs an is greater than or equal test between ‘x’ and ‘y’,
       ‘ returns 0 on false and 1 on true, works for static
       ‘ initialisation and for run time execution
x < y  ‘ performs an is less than test between ‘x’ and ‘y’,returns 0 on
       ‘ false and 1 on true, works for static initialisation and for
       ‘ run time execution
x > y  ‘ performs an is greater than test between ‘x’ and ‘y’,returns 0
       ‘ on false and 1 on true, works for static initialisation and for
       ‘ run time execution
```

Tips, Tricks and Hints

The ampersand hint, '&' and '&&'

- The compiler uses one or more **&** characters to control how branching and jumping is performed.
- When writing code you do not need to worry about page jumps and code spilling out of pages or sections, the compiler will automatically stitch all the fragmented areas of code together using page jumps and accumulator save or restore code, (***thunks***).
- You can completely ignore these compiler hints and the code will work just fine, but if you want to, (sometimes drastically), reduce your code's size and increase it's speed then you can use these hints as follows:
- Use one **&** or two **&&** hints for conditional statements and use one **&** hint for looping and jumping statements.
- No hints allows full relational operator expressions in conditional statements, i.e., **AND, OR, XOR, NOT**, (use parenthesis so that order of precedence works correctly).

```
IF (a > 50) AND (b < 20) THEN GOSUB doSomething
```

- Using one **&** hint; relational operator expressions in conditional statements will no longer work, (you will not be warned, they will result in undefined behaviour), jumping will be more efficient as it uses macros rather than subroutines, (the following example emulates the **AND** conditional statement from above with nested **IF** statements). The order of the **IF** statements in a nested block can drastically effect code execution speed, (especially in tight loops), generally the **IF** statements should be ordered in terms of chance to occur from least::outer to most::inner for best results. This a crude example of modern day Short-circuit evaluation, i.e., the second **IF** statement is not evaluated if the first **IF** statement fails.

```
IF a &> 50
    IF b &< 20 THEN GOSUB doSomething
ENDIF
```

- Using two **&&** hints, relational operator expressions in conditional statements will no longer work and code will use branches instead of jumps, but can fail if the branch destination is in another page, (the *'Validater'* will warn you if this occurs and you will then either have to re-arrange your code to make the branch and it's destination fit in one page, or the easier alternative it to use one **&** hint or no hints as above).
- Generally you should not be using two hints **&&** until the project is finished and you are trying to extract every last byte of RAM or vCPU cycles in the final optimisation phase, the other common time to use two hints **&&** is within extremely tight or inner loops.
- Two hints **&&** can only be used on conditions, not on **GOTO**, **GOSUB** or **FOR-NEXT** loops.

```
IF a &&> 50
    IF b &&< 20 THEN GOTO &doSomething
ENDIF
```

- The following **FOR-NEXT** loop uses a branch instead of a jump because of the **&** compiler hint.

```
FOR i=0 &TO 10
    PRINT i
NEXT i
```

- The following **REPEAT UNTIL** loop uses a branch instead of a jump because of the **&&** compiler hint.

```
REPEAT
    INC i
UNTIL i &&>= 10
```

- The following **REPEAT FOREVER** loop uses a branch instead of a jump because of the **&** compiler hint.

```
REPEAT
    PRINT “HELP I’M TRAPPED!”
&FOREVER
```

- The following **WHILE WEND** loop uses a branch instead of a jump because of the **&&** compiler hint.

```
WHILE i &&> 0
    i = i - 1
WEND
```

- The following **GOTO** statement uses a branch instead of a jump because of the **&** compiler hint.

```
GOTO &doStuff
```

- The following **GOSUB** statement declares that it does not require a **PUSH** instruction at the subroutine’s ‘*doSomething’* entry-point because of the **&** compiler hint. This is an advanced optimisation that is normally only used in *Vertical Blank* handlers or raw assembly code subroutines.

  Use this optimisation with caution!

```
GOSUB &doSomething
```

Variable initialisation

- Pack the initialisation of your variables into multi-statement lines if you can, your code will be smaller and faster. This will produce fewer vCPU instructions making your initialisation phase use less RAM and execute more quickly.

```
‘ use code like this
x = 0 : y = x : z = y

‘ instead of this
x = 0
y = 0
z = 0
```

- You can take the above method even further by short circuiting the normal save-restore expression result paradigm.

```
‘ use code like this when you can; the result of x is used immediately
‘ by the next calculation, without using the normal “save restore
‘ expression result” paradigm, this saves vCPU instructions, vCPU cycles
‘ and RAM.
x = y * 3 : z = x + 4

‘ instead of this
x = y * 3
z = x + 4
```

- An even better way of initialising variables is by using the **DEF** statement, this command can be used to notify the compiler of your intention to use a list of variables, but have them initialised at a later stage. This costs less code and thus saves vCPU cycles and RAM, especially as your list of variables gets bigger, (*Note* currently **INIT** can only initialise int16 variables to zero).

```
‘ notify the compiler about the existence of these variables.
DEF lives, delay, x, y

init:
    GOSUB initVars

REPEAT
    ‘ do stuff
FOREVER

initVars:
    ' initialises all variables, (to zero), starting at lives
    INIT VARS @lives
return
```

Expression handler

- Short circuit complex literal calculations into simpler real instructions. This basically means to group multiple literal or constant calculations by parentheses as much as possible. This example incorporates the multi-statement per line optimisation from above as well as reducing instructions by having the compiler calculate the literals at compile time, rather than vCPU code doing the calculations at run-time.

```
‘ do this
x = 20 : y = x*20 + (53 - 12 - 9)

‘ instead of this
x = 20
y = x*20 + 53 - 12 - 9
```

- Use as much floating point calculations with literals, (including transcendentals), as you want; it will all be calculated at full double floating point precision and then the final answer rounded down into int16_t, (the native vCPU 16 bit format).
- Remember to always use parenthesis around your complex literal calculations or you will calculate garbage.
- No variables can be used within the parenthesis containing the floating point calculations.
- Transcendentals use Degrees, not Radians.

```
‘ this produces valid code
x = 1 : y = x*2*(50*exp(-1.232)*sin(45)*cos(57.324) - 1000.346324)

‘ this does not produce valid code
x = 1 : y = x + exp(-1.232)

‘ change it to this to get correct results, (*Note* the floating point
‘ literals will be calculated at full double floating point precision
‘ and then be rounded to fit into int16.
x = 1 : y = x + (exp(-1.232))
```

Input

- There are no INKEY or GETKEY functions as it is trivial to perform any type of key or button processing you require using the **GET** function.
- You can use **GET***("SERIALRAW")* or **GET***("BUTTONSTATE")* and do your own edge or level and keyup or keydown detection.
- Use the **INPUT** command to it’s fullest potential, it offers heading strings as well as field strings, field widths with automatic field boundary checks, multiple string and int16 variable initialisation, edge of screen text scrolling, granular control over newlines with semi-colon formatting and more!
- The following is an example of how to perform rising edge detection keyboard input, that is the PRINT commands will be triggered on the down press of the keyboard keys.

```
‘ rising edge, (down press), input triggering
kk = 255

loop:
    k = GET("SERIALRAW")
    IF kk &&= 255
        IF k &&<> 255 THEN GOSUB k
    ENDIF
    kk = k
GOTO &loop

49: PRINT "1" : RETURN
50: PRINT "2" : RETURN
51: PRINT "3" : RETURN
```

Delays

- You can wait a predetermined period of time using the **WAIT** command.
- **WAIT** without any parameters waits for one vertical blank, (use this to sync your fast inner loops to vertical blank).
- **WAIT** <n> will wait 'n' vertical blank periods, where 'n' can be 1 to 32767.

```
‘ wait 1 vertical blank
WAIT

‘ wait 10 vertical blanks
WAIT 10
```

Constants

- The **CONST** statement defines the entity following it as a literal constant either taking up no memory space, (int vars), or instanced memory space, (strings). Instanced memory space for strings means that all references to that string will reference the exact same string in memory. If you require unique modifiable copies of strings, do not use **CONST** strings.
- Use constants liberally, you can use expressions to evaluate constants making code more readable and easier to understand.

```
CONST fontstart = 0
CONST boingstart = fontstart + 27
LOAD SPRITE, ../../images/gbas/Boing/Boing0.tga, boingStart + 0
LOAD SPRITE, ../../images/gbas/Boing/Boing1.tga, boingStart + 1

‘ CONST allows you to easily reference an instanced copy of a literal
‘ string anywhere in your code
CONST name$ = "insert name here"

PRINT name$
IF s$ = name$ THEN GOSUB doSomething
```

Variables

- There are **40** int16, (2 byte signed integer), fast variables available for program use.
- If you need more variables then you can use arrays, (**DIM** or **DEF BYTE *or* DEF WORD**).
- If you need byte sized variables then you can use the **DEF BYTE** statement with **POKE** and **PEEK**.
- **DEF WORD** and **DEEK** or **DOKE** can be used as faster versions of int16 arrays inside tight inner loops.

Arrays

- Array indices begin at 0, so **DIM**(n) creates n+1 array values.
- Arrays are not contained contiguously in memory, (only the first dimension is guaranteed to be contiguous), so traditional formulas to allow peeking and poking do not work, (this is because of the extreme fragmented memory architecture of the Gigatron in it's base 32K RAM configuration).
- Arrays are neither row major or column major in topology, they are contained in memory as ***k*** copies of ***j*** copies of ***i*** values for a three dimensional array, ***j*** copies of ***i*** values for a two dimensional array and ***i*** values for a one dimensional array.
- Arrays are stored as a series of indirection tables pointing to the ***k * j*** copies of memory values actually stored in memory, (for a 3 dimensional array). This means that more RAM is required to store an array than just the RAM required to store the array's values.
- Use the @ operator to get the memory address of a one dimensional array.

```
‘ get the memory address of a single dimensional int16 array
DIM oneDim(5)
a = @oneDim
```

- Use the **ADDR** function to get the memory address of any element within any type of array.

```
‘ get memory address of an element in an array
DIM twoDim(5, 5)
a = ADDR(twoDim(2, 2)
```

- Multi dimensional arrays can be slower to access.
- Strings are internally stored in the same way as a single dimensional array, (with an extra length byte and an extra delimiting zero), you can therefore reference individual characters using an array index.
- String arrays are internally stored as a two dimensional array of bytes, you can reference individual characters using array indices.

```
‘ using an array index to access a string’s characters
‘ index=0 will return the length and index=length-1 will return the
‘ terminating zero
c = str$(10)

‘ using array indices to access an array of string’s characters
‘ index0=0 will return the length and index1=length-1 will return the
‘ terminating zero, indices are ordered (1, 0)
c = strArr$(1, 10)
```

- Arrays do not use any of page zero space, so you may use as many as you like until you run out of RAM.
- Arrays can have initialiser lists stored as a long line sequence of values or within a compound initialiser list. If the initialiser list contains less entries than the total size of the array, then the remainder of the array is initialised with the last entry in the initialiser list.

```
‘ the following word array, has six elements, (0..5), that are
‘ initialised with values 0, 1, 2, -1, -1, -1
DIM a(5) = 0, 1, 2, -1

‘ the following byte array has ten elements initialised from a compound
‘ initialiser list, {}, i.e. a list that can spread over multiple lines
DIM a%(9) = {0, 1, 2, 3, 4
             5, 6, 7, 8, 9}
```

Static initialisation

DEF magic

Assembler

- The compiler always compiles and links to a list of vCPU instructions that are saved into a ***.gasm*** output file, the assembler then assembles these vCPU instructions, (mnemonics), into vCPU hex code that is then saved into a binary code ***.gt1*** file.
- This is a different order of operation to a traditional compiler because linker files, (the **gtBASIC** run-time), are not object or binary files, they are source or text files. The run-time, (which is hand-crafted vCPU assembly code), is easily modifiable without the need for extra processing-building-linking, you can make your modifications to the run-time or add new functions to it and simply re-compile your gtBASIC source code. The changes to the run-time will automatically and immediately be linked into your gtBASIC source code just before the assembly stage.
- The compiler then assembles the resultant output of your gtBASIC source code plus the run-time source code to produce a ***.gasm*** file. This output assembly file may then be viewed, (and even edited to some degree), in any text editor of your choosing; to determine how good or bad a job the compiler has performed.
- The assembly output code is fully annotated and you can directly see which gtBASIC statements have produced which vCPU instructions or mnemonics.
- The following code is a short subset of the assembled version of ‘Blinky.gbas’, you can see the labels, macros and annotated code.

```
; Code
_entryPoint_        InitRealTimeProc
                    InitEqOp
                    InitNeOp
                    InitLeOp
                    InitGeOp
                    InitLtOp
                    InitGtOp
                    Initialise
; INIT

_20                 LDI                     80
                    STW                     _X
                    LDI                     60
                    STW                     _Y
                    LDI                     0
                    STW                     _I
; X=160/2:Y=120/2:I=0

_40                 LDWI                    256
                    ADDW                    _Y
                    ADDW                    _Y
                    PEEK
                    STW                     0xc2
                    LD                      0xc2
                    ST                      giga_vAC + 1
                    ORI                     0xFF
                    XORI                    0xFF
                    ADDW                    _X
                    STW                     _P
; P=(PEEK(256+Y+Y) LSL 8)+X
```

- The assembler can be accessed through gtBASIC source code through the **ASM** statement, this powerful command allows full access to all of the facilities of the external assembler tool, ‘gtASM’.

```
‘ vCPU assembly code can be incorporated into your gtBASIC code like
‘ this
ASM
   LDWI   0xBEEF
   STW    _x
ENDASM
```

- Assembly code can be used anywhere within your gtBASIC source code, within initialisation, your main loop, subroutines and procedures, there are no limitations.
- Assembly code can access any gtBASIC variables, arrays, strings, constants, labels, etc, by prepending the name with an underscore.

```
x = 0

ASM
    LDW    _x
    ADDI   1
    STW    _x
ENDASM
```

- Assembly code can call gtBASIC subroutines and jump to gtBASIC labels.

```
‘ the main loop is all in assembly code, it loads the address of the
‘ xadd1 subroutine into the vCPU accumulator and then performs a CALL on
‘ the contents of the vCPU accumulator
‘ it finishes with a branch, (BRA), back to the label loop
‘ this version of code would run on all ROM versions
x = 0

loop:
    ASM
        LDWI   _xadd1
        CALL   giga_vAC
        BRA    _loop
    ENDASM

xadd1:
    x = x + 1
return

‘ this version of code would run only run on ROM versions >= ROMv5a
x = 0

loop:
    ASM
        CALLI  _xadd1
        BRA    _loop
    ENDASM

xadd1:
    x = x + 1
return
```

Optimiser

- The optimiser currently only spans single lines and multi statement lines, (it will be expanded in the future); so use it effectively or disable it by spreading your code out over multiple lines.
- There are many sequences of instructions that are matched-replaced-relocated by smaller more efficient sequences, check the source code for a full run down.
- Optimising causes var and/or label names to contain meaningless address values because of code relocation in the resultant ***.gasm*** files.
- Do not rely on the address values embedded within label names when reading vCPU assembly code in the resultant ***.gasm*** files, their purpose is only to provide unique names.
- When a line of gtBASIC source code uses the result of a previous line's calculations, the optimiser may be able to save LDW instructions if you spread the code over multi-line statements.

```
‘ optimiser is unable to remove a superfluous LDW because the code is
‘ spread over multiple lines
p = p + 4
q = p

‘ generates
LDW    _p
ADDI   4
STW    _p
LDW    _p
STW    _q


‘ optimiser is able to remove the superfluous LDW
p = p + 4 : q = p

‘ generates
LDW    _p
ADDI   4
STW    _p
STW    _q
```

- The optimiser will do it’s best to optimise slow array accesses where it can, but it cannot alias internal calculations when referencing multiple array accesses with the same indices. In these cases it is much more efficient to alias the index calculations yourself and use **PEEK** or **POKE** and **DEEK** or **DOKE**.

```
‘ the optimiser can not optimise or alias internal index calculations
DIM a(8)
i = 1
a(i*3) = a(i*3) + 4

‘ for efficiency you could do it like this, the ‘LSL 1’ which is an
‘ effective multiply by 2 is needed because this is an int16 array
DIM a(8)
i = 1
index = i*3 : pointer = @a+(index LSL 1) : DOKE pointer, DEEK(pointer)+4
```

Validater

- The validater will validate your code and check for a number of different exceptions or conditions that could cause your code to fail.
- It validates that the run-time version number, (*RUNTIME_VERSION),* matches between the gtBASIC run-time and the gtBASIC compiler source code.
- It validates that there are matching **PROC** commands with **CALL** commands.
- It validates **FOR NEXT** blocks, **ELSE ELSEIF** blocks, **WHILE WEND** blocks and **REPEAT UNTIL** blocks.
- It validates any code that is able to produce a BRA vCPU instruction and verifies that the BRA's destination address is contained within the same page as the BRA itself.

```
// These are the instructions and Macros that can generate BRA type
// instructions
if(opcode == "BRA")                return true;
if(opcode == "BEQ")                return true;
if(opcode == "BNE")                return true;
if(opcode == "BGE")                return true;
if(opcode == "BLE")                return true;
if(opcode == "BGT")                return true;
if(opcode == "BLT")                return true;
if(opcode == "DBNZ")               return true;
if(opcode == "%ForNextInc")        return true;
if(opcode == "%ForNextDec")        return true;
if(opcode == "%ForNextDecZero")    return true;
if(opcode == "%ForNextAdd")        return true;
if(opcode == "%ForNextSub")        return true;
if(opcode == "%ForNextVarAdd")     return true;
if(opcode == "%ForNextVarSub")     return true;
```

- It checks for code that needs to be relocated because of size constraints or because the code has been forced to move from it's current address by code preceding it.
- It inserts page jumps and thunks to stitch sequential code together that spans multiple memory pages or segments, (this is an extremely important task in a highly fragmented memory architecture such as the Gigatron's). ***Note*** thunk is a generic term used to describe small sections of code that stitch together larger sections of code, or that provide a transport mechanism between two isolated sections of code. The validater uses thunks to perform page jumps and to save and restore vAC, (the accumulator), as the page jumps can be inserted anywhere in code.
- It adjusts vCPU instruction addresses and label addresses as code is relocated.

Linker

- The gtBASIC build environment does not exactly follow a traditional compile, assemble and link sequence. gtBASIC uses a different build procedure because linker files, (the gtBASIC run-time), are not object or binary files, they are hand crafted vCPU source or text files.
- The linker knows the name of every run-time source code file and every run-time source code subroutine. These are respectively referenced as _subIncludeFiles and _internalSubs.
- It is able to link only the run-time code that is required by your gtBASIC source code from the run-time, it does this by recursively searching the _subIncludeFiles for _internalSubs that are referenced within your gtBASIC source code.
- Each _subIncludeFile represents a particular module or type of code, e.g.

```
"math.i"        ; standard math operations, mult, div, mod, etc
"memory.i"      ; memory accessing, swapping, copying, etc
"flow_control.i"; time slicing, VBlank handling, numeric GOTO/GOSUB, etc
"clear_screen.i"; clearing of memory and screen, etc
"conv_conds.i"  ; conversion of branch conditions to booleans
"graphics.i"    ; graphics functions, circle, line, rect, etc
"audio.i"       ; audio functions and MIDI
"input.i"       ; INPUT command
"print_text.i"  ; PRINT command, newline and text scrolling
"string.i"      ; string functions, left$, mid$, strcmp$, etc
"numeric.i"     ; number functions, min, sgn, clamp, BCD handling, etc
"time.i"        ; time and timer functions
```

- Each _subIncludeFile is hand crafted for efficiency, (and fun), and to each particular branch of ROM versions; currently there are three ROM branches.

```
1) ROMv1 to ROMv4    ; original vCPU instruction set
2) ROMv5a and DEVROM ; CALLI, CMPHS and CMPHU were added
3) SDROM and ROMvX0  ; CMPHS and CMPHU removed and 21 new instructions
                     ; added
```

- By specifying which ROM version you want to link to in your gtBASIC source code via pragma's, you are able to target a specific ROM target and it's feature set. Upon executing your gtBASIC code, If the target ROM does not meet your version requirement, then the run-time will flash the center pixel on the screen continuously.

```
‘ specify minimum target ROM version
_codeRomType_ ROMv3
```

- There is by necessity multiple versions of each of the _subIncludeFile files, hand crafted for each of the ROM branches and their available vCPU instructions. This provides the best possible scenario in terms of RAM usage and execution speed when compiling gtBASIC code for different ROM versions or branches.
-

```
1) ROMv1 to ROMv4    ; “math.i”
2) ROMv5a and DEVROM ; “math_ROMv5a.i”
3) SDROM and ROMvX0  ; “math_ROMvX0.i”
```