4D LABS SEMICONDUCTORS

4D Graphics Language

REFERENCE MANUAL

Document Revision: 6.3

Document Date: 26th August 2020

Table of Contents

1. 4DGL Introduction

The 4D-Labs family of embedded graphics processors (Goldelox, Picaso, PIXXI-28, PIXXI-44 and Diablo16) are powered by a highly optimized soft-core virtual engine, E.V.E. (Extensible Virtual Engine).

EVE is a proprietary, high performance virtual processor with an extensive byte-code instruction set optimized to execute compiled 4DGL programs. 4DGL (4D Graphics Language) was specifically developed from ground up for the EVE engine core. It is a high-level language which is easy to learn and simple to understand yet powerful enough to tackle many embedded graphics applications.

4DGL is a graphics-oriented language allowing rapid application development. An extensive library of graphics, text and file system functions and the ease of use of a language that combines the best elements and syntax structure of languages such as C, Basic, Pascal, etc. Programmers familiar with these languages will feel right at home with 4DGL. It includes many familiar instructions such as IF..ELSE..ENDIF, WHILE..WEND, REPEAT..UNTIL, GOSUB..ENDSUB, GOTO as well as a wealth of (chip-resident) internal functions that include SERIN, SEROUT, GFX_LINE, GFX_CIRCLE and many more.

This document covers the language style, the syntax and flow control. This document should be used in conjunction with processor specific internal functions documents, refer to Section 9 (Processor Specific Internal Functions).

2. Language Summary

This document is made up of the following sections:

- **Language Style**
  Numbers, identifiers, comments
- **Constants and Variables**
  var, var private, #constant, Inbuilt Constants, #CONST...#END, #DATA...#END
- **Pre-Processor Directives**
  #IF, #IFNOT, #ELSE, #ENDIF, EXISTS, #ERROR, #MESSAGE, #NOTICE, sizeof, argcount, #STOP, #USE .. USING, #inherit, #MODE
- **Expressions and Operators**
  :=, &(address modifier), *, +, -, *, /, %, &(as a logical operator), |, ^, ==, !=, >, <, <=, &&, ||, !, ~, <<, >>, ++, --, +=, -=, *=, /=, %=, &=, |=, ^=, ternary operators
- **Language Flow Control**
  if .. else .. endif, while .. wend, repeat .. until/forever, goto, for .. next, switch .. case
- **Functions and Subroutines**
  gosub .. endsub, func .. endfunc, return, SystemReset, ProgramExit, Argcount, @(argument pointer)
- **Processor Specific Internal Functions (Chip Resident)**
  Goldelox Internal Functions Manual
  Picaso Internal Functions Manual
  Pixxi Internal Functions Manual (For both PIXXI-28 and PIXXI-44)
  Diablo16 Internal Functions Manual

3. Language Style

4DGL is a case sensitive language. The colour of the text, in the Workshop4 IDE, reveals if the syntax will be accepted by the compiler.

3.1. Numbers

Numbers may be defined as decimal, hex or binary.

```
0xAA55          // hex number
-1234                   // decimal number
0b10011001              // binary number
```

3.2. Identifiers

Identifiers are names used for referencing variables, constants, functions and subroutines.

A valid identifier: -

- Must begin with a letter of the English alphabet or possibly the underscore (_)
- Consists of alphanumeric characters and the underscore (_)
- May not contain special characters: **~ ! @ # $ % ^ & * ( ) ` -= { } [ ] : " ; ' < > ? , . / |**

Elements ignored by the compiler include spaces, new lines, and tabs. All these elements are collectively known as the “white space”. White space serves only to make the code more legible –it does not affect the actual compiling.

Note that an identifier for a subroutine must have a colon (:) appended to it. Subroutine names are scoped locally inside functions, they are not accessible outside the function.

```
mysub:
     [ statements ]
endsub;
```

3.3. Comments

A comment is a line or paragraph of text in a program file such that that line or paragraph is not considered when the compiler is processing the code of the file.

To write a comment on one line, type two forward slashes **//** and type the comment. Anything on the right side of both forward slashes will not be read by the compiler.

Single line comment example:

```
// This is my comment
```

Also you can comment in multiple lines by enclosing between **/***and ***/**. See example.

Multi-Line comment example:

```
/*
This is a multi line
comment which can
span over many lines
*/
```

Also you can use multiple line comment in one line like:

```
/* This is my one line comment */
```

It is good to comment your code so that you or anybody else can later understand it. Also, comments are useful to comment out sections of the code, so it can be left as a reminder about a modification that was made and perhaps be modified or repaired later. Comments should give meaningful information on what the program is doing. Comment such as 'Set output 4' fails to state the purpose of the instruction. Something like 'Turn Talk LED ON' is much more useful.

3.4. Redefining Pre-Processor Directives

It is possible to add your own 'flavour' to the pre-processor by using the '$' (substitution directive).

For example, if you wanted to make your code look 'more C like' you can do the following,

```
Example:
//define some user preferences to make things look more 'C' like
#constant enum    $#constant          /* define the enum word*/
#constant #define $#constant          /* define the #define word*/
#constant #ifdef  $#IF EXISTS         /* define the #ifdef word */
#constant #ifndef $#IFNOT EXISTS      /* define the #ifndef word */
#constant #else   $#ELSE              /* define the #else word */
#constant #endif  $#ENDIF             /* define the #endif word */
#constant #include $#inherit          /* define the #include word */
#constant #if $#IF                    /* define the #if word */
```

Now, the compiler will use these new works as aliases to the default directives.

4. Variables and Constants

4.1. Variables

Like most programming languages, **4DGL** is able to use and process named variables and their contents. Variables are simply names used to refer to some location in memory - a location that holds a value with which we are working with. Variables used by the -GFX based target platforms are signed 16-bit. Variables are defined with the **var** statement, and are visible globally if placed outside a function (scope is global) or private if placed inside a function (scope is local).

Type	Resolution	Range
Integer	Signed 16 bit	-32,768 to 32,767

```
Example:
var ball_x, ball_y, ball_r;
var ball_colour;
var xdir, var ydir;
```

Variables can also be an array such as:

```
var PlotInfoX[100], PlotInfoY[100];   // Index starting from 0 to 99.
```

Global and local variables can be initialised when they are declared and are initialised to 0 by default. Local arrays are supported; however, they must be used with caution due to stack size limitations, especially on Goldelox. A variable or array can only be initialised with a constant value.

```
Example:
var myvar;            // initialise to 0 when declared
var myvar2 := 100;    // initialise to 100 when declared
var myArray[4] := [5,200,500,2000]; // initialise the array
```

You can also size an array by setting certain initialization values,

```
var myArray[] := [1,2,3,4]; // array is sized to 4
```

And, partial initializing,

```
var myBuffer[10] := [0xAA, 0x55]; // create buffer, initialize 2 entries
```

Note that arrays can only have a single dimension. Variables can also hold a pointer to a function or pointer to other variables including arrays. Also, the index starts from 0.

```
Example:
var buffer[30];       // general purpose buffer for up to 60 bytes
var buffer2[30];      // general purpose buffer for up to 60 bytes

func main()
   buffer[0] := 'AB'; // put some characters into the buffer
   buffer[1] := 'CD';
   buffer2[0] := 'EF';
   //buffer is 16bits per location so chars are packed

   buffer2[1] := 0;
   :
   :
endfunc
```

```
Example:
var funclist[2];

//---------------------------------------------------------------
func foo()
   [...some code here...]
Endfunc
//---------------------------------------------------------------
func baa()
   [...some code here...]
endfunc
//---------------------------------------------------------------
func main()
   //load the function pointers into an array
   funclist[0] := foo;
   funclist[1]: = baa;

   funclist[0]();     // execute foo
endfunc
//---------------------------------------------------------------
```

Notes concerning variables:

- Global variables and arrays are **persistent** and exist during the entire execution of a program.
- Global variables and arrays are visible to all functions and are considered a shared resource.
- Local variables are created on the stack and are only visible inside a function call.
- Local variables are released at the end of a function call.
- More data types will be added in future releases.

4.2. Private Variables

In the normal course of program execution, any variables declared locally in functions are discarded when the function exits, freeing up the stack.

If we want the value of *a* variable or variable array to be retained after the function has executed, we simply declare the variable as private.

A private variable may be initialised just like a normal global or local variable, however, it will only be initialised once during the program initialisation in the start-up code (just like a global variable); therefore, each time the function is called, the value that was set during initialisation is persistent.

```
Example:
func myfunc()
   var private hitcounter := 100; // initial hitcounter value is 100
   print("\nHits = ",hitcounter++);
endfunc
```

In the above example, each time the function is called, "hitcounter" will be printed then incremented showing the number of times the function is called starting from 100.

As stated above, Private variables are not built temporarily on the stack like normal local variables, they consume space in the global memory space. Private variables can be accessed from any other function by prepending the function name, followed by a period, to the private variable within that function.

```
Example:
func anotherfunc()
   // code........
   myfunc.hitcounter := 50;  // set hitcounter in myfunc() to 50
   // code.....
endfunc
```

4.3. Constants

A constant is a data value that cannot be changed during run-time. A constant can be declared as a single line entry with the **#constant** directive. A block of constants can be declared with the **#CONST** and **#END** directives. Every constant is declared with a unique name which must be a valid identifier.

It is a good practice to write constant names in uppercase. The constant's value can be expressed as decimal, binary, hex or string. If a constant's value is prepended with a **$**it becomes a complete text substitution with no validation during pre-processing.

```
Syntax:
#constant NAME value
or
#constant NAME1 := value1, NAME2 := value2,..., NAMEN := valueN
or
#constant NAME equation
or
#constant NAME $TextSubstitution
or
#CONST
   BUTTONCOLOUR 0xC0C0
   SLIDERMAX    200
#END

#constant : The required symbol to define.
NAME : The name of the constant
value : A value to set the symbol to
```

```
Example:
#constant GO_FLAG 16
#constant LOGON $Welcome

//in the function...
func main()
   var flags;
   flags := 16;
   if (flags && GO_FLAG)
      putstr (LOGON);
      run_process();
   endif
   :
endfunc
```

Note also that the := operator can be included for readability. For example,

```
#constant BUFSIZE1 := 10
#constant BUFSIZE2 := 20
#constant TOTALBUF := BUFSIZE1+BUFSIZE2
#constant BUF_SAFE_LIMIT  := (BUFSIZE1+BUFSIZE2) * 750 / 1000
// 75% mark

#NOTICE "Buffer safe limit = ",BUF_SAFE_LIMIT
// show in Workshop4 error window
```

Constants can now be automatically enumerated,

```
#constant Hearts, Diamonds, Clubs, Spades
```

Each of the enumeration values is equated with a sequential number. The enumeration will have the following values assigned. Hearts = 0, Diamonds = 1, Clubs = 2, Spades = 3

You can also override the default number assignments. For example,

```
#constant Hearts:=10, Diamonds, Clubs:=20, Spades
//now, Hearts = 10 , Diamonds = 11, Clubs = 20, Spades = 21
```

#constant must have all the definitions on the same line and traditionally is only used for single constant initialisation. #CONST blocks are usually more readable for multiple constant assignments. For example,

```
#CONST
  Hearts:=10,
  Diamonds,
  Clubs:=20,
  Spades,
  Total:= Hearts+Diamonds+Clubs+Spades
#END
#NOTICE "Total = ", Total  // show result of constant 'Total'
```

4.4. Inbuilt Constants

The compiler now checks the program and memory sizes and reports error if amount is exceeded. This used to be done by Workshop4. The following constants must be defined in the platforms fnc file for this to work. For example,

```
Example:
#constant __MAXMEM    255
#constant __MAXPROG   9216
```

To allow the programmer to recognise what platform is being used, the device is also described with a constant in the fnc file. For Goldelox,

```
//in the GOLDELOX.fnc file we have
#constant GOLDELOX    1
#constant __PLATFORM GOLDELOX
```

This allows the programmer to make coding decisions,

```
#IF __PLATFORM == GOLDELOX
// code it this way
#ELSE
// code it that way
#ENDIF
```

4.5. Data Blocks: #DATA … #END

#DATA blocks reside in the CODE space and can be **bytes** or **words**. Data cannot be changed during run-time (i.e. it is read only). A block of data can be indexed like an array. A block of data is declared with the #DATA and #END directives. Every data entry is declared with a unique name which must be a valid identifier.

```
Syntax:
#DATA
   type name
   value1, value2, .... valueN
#END

or

#DATA
   type name  value1, value2, .... valueN
#END

or

#DATA
   type name equation1, equation2,....equationN
#END

#DATA : The required symbol to define.
type     : byte or word data type keyword
name     : The name of the data array
value : A list of 8 bit or 16 bit values in the data array
```

```
Example1:
#DATA
   word values
   0x0123, 0x4567, 0x89AB, 0xCDEF
   byte hexval
   "0123456789ABCDEF"
#END
//and in a function,
func main()
   var ch, wd, index1, index2;
   ch := hexval[index1];
   // load ch with the correct ascii character

   wd := values[index2];   // get the required value to wd
   :
endfunc
```

#DATA statements can now contain pre-processor computed values. For example,

```
Example2:
  #constant SPEED 33
  #DATA
    word mylimits (BUFSIZE1+BUFSIZE2)*750/1000, 100, 200, (SPEED*20)
  #END

  func main()
    var n;
    for(n:=0; n < sizeof(mylimits); n++)  print(mylimits[n],"\n");
  endfunc
```

#DATA statements can reference functions and variables. For example,

```
Example3:
#DATA word myfuncs tom, dick, harry

func main()
   var n;
   for(n:=0; n < sizeof(myfuncs); n++)  myfuncs[n]();
endfunc

func tom() putstr("Tom\n"); endfunc
func dick() putstr("Dick\n"); endfunc
func harry() putstr("Harry\n"); endfunc
```

5. Pre-processor Directives

5.1. #IF, #IFNOT, #ELSE, #ENDIF, EXISTS, #ERROR, #MESSAGE, #NOTICE, sizeof, argcount

These pre-processor directives allow you to test the value of an arithmetic expression, or the existence of a pre-defined constant or PmmC function name, or the size of a predefined array. #IFNOT gives the inverse response of #IF. Combined with #ELSE and #ENDIF, conditional inheritance of other files and control of blocks of code can be performed.

Note: If you wish to add a comment following a preprocessor definition, the preferred style is /*.....*/. The // comment style will not work in many cases as it breaks the 'inline' rule by ignoring everything to the end of line following the // comment directive.

Example:

```
#constant NUMBER  10000
var buffer[30];
#MESSAGE "Example Code"   // messages go to the *.aux file
#IF (__PLATFORM == GOLDELOX) & (NUMBER == 10000)
#NOTICE "Compiling for Goldelox Project ", NUMBER, ", Buffer
size is ",sizeof(buffer)
#ELSE
#ERROR "Unknown Platform"
#ENDIF

// check for existence of a PmmC function
#IF EXISTS gfx_Rectangle
#NOTICE "we have gfx_Rectangle, token value = ", gfx_Rectangle
#NOTICE "gfx_Rectangle requires ", argcount(gfx_Rectangle), "
arguments"
#ELSE
func gfx_Rectangle(var x1,var y1,var x2, var y2, var colour)
// code your own function here
endfunc
#ENDIF
```

5.2. #STOP

#STOP terminates compilation, it is mainly used for debugging purposes to disable compilation of code from a certain point onwards.

Example:

```
#STOP //compiler will stop here
```

5.3. #USE, USING

The compiler will compile all functions it finds, but will only link functions that are referenced. This allows to inherit other files that could be considered as function libraries with your favourite routines in them.

To reduce compile time and improve code readability, the #USE and USING preprocessor directives are used to selectively include functions in #inherited files, allowing you to build libraries of code, but only selectively inherit the functions required by the project.

```
Example:
//In a file named mylib.lib
#IF USING tom
  func tom() putstr("Tom\n"); endfunc
#ENDIF
#IF USING dick
  func dick()
  putstr("Dick\n");
  endfunc
#ENDIF

#IF USING harry
  func harry() putstr("Harry\n"); endfunc
#ENDIF
#IF USING george
  func george() putstr("George\n"); endfunc
#ENDIF
//=======================================
//now in your main file
#USE tom, george

//this instructs the compiler to compile those functions and use
//them in your program

func main()
tom();
george();
endfunc
// note that dick and harry are not compiled or linked
```

5.4. #inherit

#inherit "*filename*" includes another source file. Inherited files can be nested. Inherited files can now also contain #DATA, functions and variables. There is no special ordering required for functions, func main can now be the first function in the program if required, however, preprocessor calculation cannot be forward referenced. There is now also a mechanism to allow selected portions of an inherited file to be compiled on demand (see #USE and USING below).

```
Syntax:
#inherit "filename"

filename:".fnc" file to be included in the program.
```

```
Example:
#platform "GOLDELOX"
#inherit "4DGL_16bitColours.fnc"

func main()
:
:
endfunc
```

5.5. #MODE

#MODE is used to define Pre-Processor Directives such as the memory section you want you program to run from, and Flash Memory protection. These Pre-processor Directives do not apply to Goldelox.

#MODE is defined at the start of your program, typically after any #inherit lines.

Picaso, Pixxi-28 and Pixxi-44 Processors have the following directives available

Directive	Description
RUNFLASH	Program will execute from processor Flash Memory. If omitted, the Program loaded in Flash memory will load to RAM before it is run. A program that runs directly from Flash will run a little slower than a program that is run from RAM but has the advantage of leaving RAM space free for other functions.

Diablo16 has the following directives available

Directive	Description
RUNFLASH	Irrelevant for Diablo as the RUNFLASH overhead for Diablo is negligible and therefore is always used. The Program will execute from processor Flash Memory whether this is defined or not. Listed for information purposes only.
FLASHBANK_0	Program target is Flashbank 0 (or 1, 2, 3, 4, 5 as defined). This is useful when writing child programs and defining which Flashbank the child program will reside in.
FLASHBANK_1	
FLASHBANK_2	
FLASHBANK_3	
FLASHBANK_4	
FLASHBANK_5	
SAVE_TO_DISK	This saves the current program to the uSD on the display. Ensure that the program file on your computer is saved and has an 8.3 filename, otherwise the first 12 characters will become the filename.ext
FLASH_READ_PROTECT	This makes the Flashbank that the code is written to read protected so that it cannot be read by flash read type commands.
FLASH_WRITE_PROTECT	This makes the Flashbank that the code is written to write protected so it cannot be overwritten using the various flash write type commands. It can only be erased using flash_EraseBank() with the confirmation.

If multiple #MODE directives are required together, such as Flashbank with Read and/or Write protection, use the '+' symbol between directives on a single #MODE line.

```
Syntax examples:

#MODE RUNFLASH // This prog intended to be 'front end' and run from FLASH

or

#MODE SAVE_TO_DISK // This prog will be saved to uSD card on Diablo

or

// This prog will reside on FlashBank 2 and will be Read/Write Protected
#MODE FLASHBANK_2 + FLASH_READ_PROTECT + FLASH_WRITE_PROTECT
```

5.6. #STACK

The Stack is a fixed portion of RAM on the system Heap (The Heap comprises all the RAM available to the user and is largely managed automatically) when a program starts. It contains local variables, parameters, and the return address of the previous function for the current function. The same information for all previous functions is also stored on the stack.

#STACK xxx is used to define the size of RAM, where xxx is the desired size. The default value, when not specified is 200 'entries' in size, each entry is a word long.

In ViSi-Genie, the #STACK is handled automatically.
In Designer and ViSi, the #STACK needs to be handled by the User, as the size of the Stack required depends on the code that has been written. Generally, the default is adequate, although for programs with recursive functions, functions with large quantities of variables, or where the function 'depth' is large, a larger value might be needed.

The compiler produces a STACK_ESTIMATE value in the .aux file, this value assumes every function calls every other function and there is no recursion, so it is only useful as a guide. After compiling, the .aux file can be opened in a text editor and the STACK_ESTIMATE value can be seen. Remember though, this is a guide only and may be insufficient for your application.

Syntax:
```
#STACK 500
```

If the #STACK value is too small, this will result in an 'EVE Stack Overflow' error. Too large, and you simply waste RAM and thus could result in your program running out and causing other errors or undesirable effects.

6. Expressions and Operators

Operators make 4DGL a powerful language. An operator is a function which is applied to values to give a result. These operators take one or more values and perform a useful operation. The operators could be +, -, & etc.

Most common ones are arithmetic operators. Other operators are used for comparison of values, combination of logical states and manipulation of individual binary digits.

Expressions are a combination of operators and values. The values produced by these expressions can be used as part of even larger expressions or they can be stored in variables.

6.1. Assignment Operator

6.1.1. ":=" Assign a value to a variable:

```
Example:
a := b + c*(d-e)/e;
a := b + 22;
```

6.2. Address Modifiers

6.2.1. "&" Get the address of a variable:

```
Example:
a := &myArray[0];
a := myArray;    // same as above. The array name without indices
a := &b;         // implies an address, same as in the C language.
```

6.2.2. "*" Use a variable as a pointer:

```
Example:
a := *b;              // Can be used on the right side and/or
*j := myArray;        // left side and/or
*d := *s;             // even both sides of assignment operator.
```

Note: When using a variable as a pointer, it is a good practice to declare it as such. Otherwise, the compiler will generate a notice. To illustrate, below is an example of a variable being used as a pointer, but not declared as such.

```
Example:
var a; // a is declared as a variable
var b; // b is a variable
var j[3] := [5,200,500]; // j is an array with initial values

a := j; // a holds the address of j
b := a[0]; // b is equal to the first element of array j

print ( "j[0]: ", b,"\n");
```

The example would compile and print “j[0] : 5” on the display module. However the message area would display the notice below.

```
Notice: variable 'a' is being indexed (line 12 file:pointerTest2.4dg)
0 errors
0 warnings
1 notice
No Errors, code size = 84 bytes out of 14400 total
Initial RAM size = 200 bytes out of 14400 total
Program will run from ram so total initial RAM size = 284 bytes out of 14400 total
Download to Flash successful.
```

Hence the correct way is to declare “a” explicitly as a pointer.

```
Example:
var *a; // a is declared as a pointer
var b; // b is a variable
var j[3] := [5,200,500]; //j is an array with initial values

a := j; // a is a pointer which holds the address of j
b := a[0]; // b is equal to the first element of array j

print ( "j[0]: ", b,"\n");
```

The compiler notice is now removed.

```
0 errors
0 warnings
0 notices
No Errors, code size = 84 bytes out of 14400 total
Initial RAM size = 200 bytes out of 14400 total
Program will run from ram so total initial RAM size = 284 bytes out of 14400 total
```

6.3. Arithmetic Operators

6.3.1. "+" Addition:

```
Example:
val1 := 5;
val2 := 10;
sum  := val1 + val2;
```

6.3.2. "-" Subtraction:

```
Example:
val1 := 5;
val2 := 10;
diff := val2 - val1;
```

6.3.3. "*" Multiplication:

```
Example:
velocity := 5;
time := 10;
displacement := velocity * time;
```

Goldelox: the overflow (bits 16 to 31) is placed into the **VM_OVERFLOW** register which can be read by the **OVF()** function.

6.3.4. "/" Division:

```
Example:
delta := 5;
length := 10;
strain := delta/length;
```

Goldelox: the remainder is placed into the **VM_OVERFLOW** register which can be read by the **OVF()** function

6.3.5. "%" Modulus:

```
Example:
a := 3;
b := 11;
remainder := b%a;  // remainder is 2
```

*, / and % have the higher precedence and the operation will be performed before + or – in any expression. Brackets should be used to enforce a different order of evaluation. Where division is performed between two integers, the result will be an integer, with the remainder discarded. If a program is ever required to divide a number by zero, this will cause an error, usually causing the program to crash.

6.4. Comparison Operators

4DGL has set of logical operators useful for performing comparisons. These operators all return a TRUE (1) or FALSE (0) depending on the result of the comparison. These comparisons are most frequently used to control an 'if' statement or a repeat or a while loop. Note that '==' is used in comparisons and ':=' is used in assignments.

6.4.1. "==" Equals test:

```
Example:
if(index == count)
        ..
        print("count complete");
endif
```

6.4.2. "!=" Not Equals test:

```
Example:
if(denominator != 0)
        ..
   result := numerator/denominator;
endif
```

6.4.3. ">" Greater Than test:

```
Example:
while(index > 0)
        ..
   index--;
wend
```

6.4.4. ">=" Greater Than or Equals test:

```
Example:
while(index >= 0)
        ..
   index--;
wend
```

6.4.5. "<" Less Than test:

```
Example:
while(index < 10)
   ..
   index++;
wend
```

6.4.6. "<=" Less Than or Equal:

```
Example:
while(index <= 10)
   ..
   index++;
wend
```

6.5. Boolean Logical Operators

4DGL provides common logical operators designed to return TRUE (1) or FALSE (0) depending on the result of the expression. These operators both return boolean results and take boolean values as operands.

6.5.1. "&&" Logical AND test:

```
Example:
if(x < 10 && x > 5)  print("x within range");
```

6.5.2. "||" Logical OR test:

```
Example:
if(x > 10 || x < 5)  print("x out of range");
```

6.5.3. "^" Logical XOR test:

```
Example:
if ((A< B) ^ (C < D))
   putstr("Expression is true");
endif
```

6.5.4. "!" Unary NOT:

The unary NOT operator sets a 1 if all bits in a variable are zero, otherwise sets 0

```
Example1:
  while(!x)
        .....  continue operation while x == 0
  wend
```

The NOT (!) operator inverts the boolean result of a value, or the result of an expression. For example, if a variable named *flag* is currently 55, prefixing the variable with a '!' character will make the result FALSE (0).

```
Example2:
var flag := 55; //variable is non zero
var flag2;
flag2 := !flag; // flag2 set to FALSE (0)
flag2 := !flag2; // now flag2 set to TRUE (1)
```

6.5.5. "~" Unary 2's COMPLEMENT:

The unary 2's COMPLEMENT (~) operator inverts all bits

```
Example:
  x := 0x5555;
  print([HEX]~x); //Prints 0xAAAA
```

6.6. Bitwise Operators

6.6.1. "&" Bitwise AND:

The Bitwise AND is represented by a single ampersand (&). It makes a bit by bit comparison of two numbers. Any corresponding position in the binary sequence of each number where both bits are 1 results in a 1 appearing in the same position of the resulting number. If either bit position contains a 0 then a zero appears in the result.

```
Example:
var j := 0b0000000010101011; var k := 3; var r;
r := k & j; // Perform bitwise AND on variables k and j
print("The result is ", r); // Result is 3
```

6.6.2. "|" Bitwise OR:

The bitwise OR performs a bit by bit comparison of two binary numbers. The OR operator places a 1 in the result if there is a 1 in the first or second operand.

```
Example:
var j := 0b0000000010101011;
var k := 3;
var r;
r := k | j; // Perform bitwise OR on variables k and j
print("The result is ", r); // Result is 171
```

6.6.3. "^" Bitwise XOR:

The bitwise XOR sets a 1 if one or other corresponding bit positions in the two numbers is 1. If both positions are a 1 or a 0 then the corresponding bit in the result is set to a 0.

```
Example:
var j := 0b10101011;
var k := 3;
var r;
r := k ^ j; // Perform bitwise OR on variables k and j
print("The result is ", r); // Result is 168
```

6.6.4. "<<" Shift Left:

The bitwise left shift moves each bit in a binary number a specified number of positions to the left. As the bits are shifted to the left, zeros are placed in the vacated rightmost (low order) positions. Note that once the left most (high order) bits are shifted beyond the size of the variable containing the value, those high bits are shifted into the VM_OVERFLOW register and can be read with OVF() function. VM_OVERFLOW can also be read using peekW(VM_OVERFLOW) on the Goldelox processor only).

```
Example:
var k := 0b0110111100000000;  // 28416
var r;
pokeW(VM_OVERFLOW, 0);       // clear the overflow register
r := k << 3;          // shift k 3 bit positions to the left
print("The result is ", r, "OVF() = ", OVF());
```

The example code would display the shifted result, which is 3552 and the overflow value is 3.

The VM_OVERFLOW register is not cleared prior to a shift, this allows you to do interesting things such as rotating an array, The VM_OVERFLOW register must be cleared (or pre-set to a required value) prior to using the shift instruction if you wish to obtain the correct result.

Goldelox: the most significant bit goes out and into the **VM_OVERFLOW** register which can be read by **OVF()** function.

6.6.5. ">>" Shift Right

A bitwise right shift is much the same as a left shift except that the shift tales place in the opposite direction. Note that the low order bits that are shifted off to the right are shifted into the VM_OVERFLOW register and can be read with OVF() function (VM_OVERFLOW can also be read using peekW(VM_OVERFLOW);) the vacated high order bit(s) position are replaced with zeros.

```
Example:
var k := 0b0000000010101011;  // 171 var r;
pokeW(VM_OVERFLOW, 0);       // clear the overflow register
r := k >> 3;          // shift k 3 bit positions to the right
print("The result is ", r, "OVF() = ", OVF());
```

The above code would display the shifted result, which is 21 and the overflow value is 16384.

The VM_OVERFLOW register is not cleared prior to a shift, this allows you to do interesting things such as rotating an array. The VM_OVERFLOW register must be cleared (or pre-set to a required value) prior to using the shift instruction if you wish to obtain the correct result.

Goldelox: the least significant bit goes out and into the **VM_OVERFLOW** register which can be read by **OVF()** function.

6.7. Short Hand Notations

The iterator(..); function can be used to modify the post or pre increment value for the next execution of the Post/Pre increment and decrement operation. The increment value is automatically set back to 1 after the next Post/Pre increment and decrement operation.

6.7.1. "var++" Post-Increment:

```
Example:
x := a*b++;
// equivalent to
x := a*b;
b := b+1;
```

6.7.2. "++var" Pre-Increment:

```
Example:
x := ++b*a;
// equivalent to
b := b+1;
x := a*b;
```

6.7.3. "var--" Post-Decrement

```
Example:
x := a*b--;
// equivalent to
x := a*b;
b := b-1;
```

6.7.4. "--var" Pre-Decrement

```
Example:
x := --b*a;
// equivalent to
b := b-11;
x := a*b;
```

```
Example:
//Using iterator
iterator(10);
myarray[x++] := 123;

// equivalent to
myarray[x] := 123;
x := x + 10;
```

6.8. Compound Assignment Operators

4DGL now provides a number of operators designed to combine an assignment with a mathematical or logical operation. These are primarily of use when performing an evaluation where the result is to be stored in one of the operands. For example, you may write an expression as follows:

6.8.1. += Operator

```
Example:
k += j; //Add j to k and place result in k
```

6.8.2. -= Operator

```
Example:
k -= j; //Subtract j from k and place result in k
```

6.8.3. *= Operator

```
Example:
k *= j; //Multiply k by j and place result in k
```

6.8.4. /= Operator

```
Example:
k /= j; //Divide k by j and place result in k
```

6.8.5. %= Operator

```
Example:
k %= j; //Perform Modulo of j on k and place result in k
```

Note: Any overflow situation from the math operators can be obtained by reading the VM_OVERFLOW register. It can be read with OVF() function. VM_OVERFLOW can also be read using peekW(VM_OVERFLOW).

6.8.6. Compound Bitwise Operators

Like the arithmetic operators, each bitwise operator has a corresponding compound operator that allows the operation and assignment to be performed using a single operator.

6.8.7. "&=" Compound Bitwise AND

```
Example:
k &= j
//Perform a bitwise AND of k and j and assign result to k
```

6.8.8. "|=" Compound Bitwise OR

```
Example:
k |= j
//Perform a bitwise OR of k and j and assign result to k
```

6.8.9. "^=" Compound Bitwise XOR

```
Example:
k ^= j
//Perform a bitwise XOR of k and j and assign result to k
```

Note: Any overflow situation from the math operators can be obtained by reading the VM_OVERFLOW register. It can be read with OVF() function (VM_OVERFLOW can also be read using peekW(VM_OVERFLOW).

6.9. Ternary Operator

4DGL now includes the ternary *operator*, a shortcut way of making decisions.

```
Syntax:

[condition] ? [true expression] : [false expression]
```

The way this usually works is that *[condition]* is replaced with an expression that will return either TRUE (1) or FALSE(0). If the result is true then the expression that replaces the *[true expression]* is evaluated. Conversely, if the result was *false* then the *[false expression]* is evaluated.

```
Example:
var k:=20, j:=40;
var r;
r := (k > j) ? k : j ;
print("Larger number is ", r);
```

The above code example will evaluate whether k is greater than j, this will evaluate to false resulting in j being returned to r.

```
print( (RAND()<10000) ? 1 : 0);
```

The above code example will print a '1' if the random number is < 1000, else it will print a '0'.

```
print([STR] (RAND()<10000) ? "low " : "high");
```

The above code example will print “low” ' if the random number is < 1000, else it will print “high”

```
(n >= 45 && n <= 55) ? myfunc1() : myfunc2();
```

The above code example will call myfunc1() if the number lies between 45 and 55 inclusive, else it will call myfunc2(). The functions may have arguments if required.

Any number of expressions may occur to the left and right of the colon (:), however, only the right-most comma separated element will actually return a value, for example, in the following code,

```
r := (n >= 45 && n <= 55) ? myfunc1() : myfunc2(), X++;
```

myfunc1() will be called if the number lies between 45 and 55 inclusive and r will receive its return value, else myfunc2() will be called, but actually returns the value of X (X is then incremented).

The return value (if any) from myfunc2() is discarded.

7. Language Flow Control

7.1. if … else … endif

The **if-else** statement is a two-way decision statement. The **if** statement answers the question, "*Is this true or false?*", then proceeds on some action based on this. If the condition was **true** then the statement(s) following the **if** is executed and if the condition was false then the statement(s) following the **else** is executed.

```
Syntax:

if(condition)
   [statements]
else
   [statements]
endif
or
if(condition) statement;
or
if(condition) statement; else statement;

condition : Required conditional expression to evaluate.
statements: Optional block of statements or single statement to be
            executed.
statement : Optional single statement.
```

```
Example:
func collision()
   if(ball_x <= LEFTWALL)
      ball_x := LEFTWALL;
      ball_colour := LEFTCOLOUR;
      xdir := -xdir;
   endif

   if(ball_x >= RIGHTWALL)
      ball_x := RIGHTWALL;
      ball_colour := RIGHTCOLOUR;
      xdir := -xdir;
   endif
endfunc
```

7.2. while … wend

Loop through a block of statements while a specified condition is true. Note that the **while** statement may be used on a single line without the **wend** statement. The **while … wend** loop evaluates an expression before executing the code up to the end of the block that is marked with a **wend**. If the expression evaluates to FALSE on the first check then the code is not executed.

```
Syntax:

while(condition)
   [statements]
   [break;]
   [continue;]
wend
or
while(condition) statement;
or
while(conditional statement);

condition : Required condition to evaluate each time through the
 loop. The loop will be executed while the condition is true.
statement : Optional block of statements to execute.
wend : Required keyword to specify the end of the while loop.

Related statements (only if in block mode):
break; : Break out of the while loop by jumping to the statement following
the wend keyword (optional).
continue; : Skip back to beginning of the while loop and re-evaluate the
condition (optional).
```

```
Example:
i := 0;
val := 5;

while(i < val)
   myVar := myVar * i;
   i++;
wend
```

```
Example:
i := 0;
val := 5;

while(i < val)  myVar := myVar * i++;
```

Example:

```
//This example shows the nesting of the while..wend loops
var rad, color, counter;
func main()
   color := 0xF0F0;
   gfx_Set(0, 1);    // set PenSize to 1 for outline objects
   while(counter++ != 1000)
      rad := 10;
      while(rad < 100)
         gfx_Circle(120, 160, rad++, color++);
         gfx_Ellipse(120, 160, rad++, 20, color++);
         gfx_Line(120, 160, 20, rad++, color++);
         gfx_Rectangle(10, 10, rad++, rad++, color++);
      wend
   wend
endfunc
```

7.3. repeat … until/forever

Loop through a block of statements until a specified condition is **true**. The statement block will always execute at least once even if the **until**(condition) result is **true**. For example, you may need to step through an array until a specific item is found.

The **repeat** statement may also be used on a single line with **until**(condition);

```
Syntax:

repeat
   [statements]
   [break;]
   [continue;]
until(condition);
or
repeat
   [statements]
   [break;]
   [continue;]
forever
or
repeat (statement); until(condition);

condition : Required condition to evaluate at the end of loop.
The loop will be repeated until the condition is true.
statement : Optional block of statements to execute.
until : Required keyword to specify the end of the repeat loop.

Related statements (only if in block mode):
break; : Break out of the repeat loop by jumping to the statement
following the until(condition) or forever keywords (optional).
continue; : Skip back to beginning of the repeat loop (optional).
```

```
Example:
i := 0;
repeat
   myVar := myVar * i;
   i++;
until(i >= 5);
```

```
Example:
i := 0;
repeat
   myVar := myVar * i;
   i++;
   if(i >= 5) break;
forever
```

```
Example:
i := 0;
repeat i++;  until(i >= 5);
```

7.4. goto

The **goto** instruction will force a jump to the label and continue execution from the statement following the label. Unlike the **gosub** there is no return. It is strongly recommended that '**goto**' **SHOULD NEVER BE USED**, however, it is included in the language for completeness.

The goto statement can only be used within a function and all labels are private within the function body. All labels must be followed by a colon '**:**'.

```
Syntax:

   goto label; // branches to the statements at label:
   ...
label:
   [statements]
   [statements]

label : Required label for the goto jump
```

```
Example:
func main()
   if(x<20)  goto bypass1;
   print("X too small");
   x := 20;
bypass1:
   if(y<50)  goto bypass2;
   print("Y too small");
   y := 50;
bypass2:
   // more code here
endfunc
```

7.5. for … next

Repeats a set of statement certain number of times.

It is possible to do things like x++, x := x + 10, or even x := random (5) and if you really wanted to, you could call other functions that do nothing to the variable but still have a useful effect on the code. Notice that a semicolon separates each of these sections. That is important. Also note that every single one of the sections may be empty, though the semicolons still have to be there. If the condition is empty, it is evaluated as true and the loop will repeat until something else stops it, like a break statement within the loop body.

```
Syntax:

for (variable initialisation; condition; variable update )
// statements to execute while the condition is true
next

variable initialization : Assign a value to an already existing
variable. More than one variable may be initialized here,
not necessarily associated with the loop control.

Condition : condition part tells the program that while
the conditional expression is true the loop should continue to
repeat itself.

Variable update : The amount by which variable is changed
each time through the loop.

Statement : Statements to execute while condition is true.

Next : Terminates the definition of the For/Next loop.

Related statements (only if in block mode):

break; : Break out of the For/Next loop by jumping to the
statement following the "Next".

continue; : Skip back to beginning of the For/Next updater (
last part).
```

```
Example:
func main()
   for (rad:=0; rad<100; rad++)
        gfx_Circle(63, 63, rad, color);
   next
endfunc
```

If a **for** loop is on a single line, 'next' may be omitted,

```
for (n:=5; n<15; n++) myfunc();
```

7.6. switch case

Runs one of several groups of statements, depending on the value of an expression. In 4DGL, there are 2 different classes of **switch-case** statement.

7.6.1. Switch with Expression

The first type is '**switch with expression**'. It evaluates the conditional expression following the **switch** statement and tests it against numerous constant values (the cases). A case that matches the value of the expression executes the statement block

```
Syntax:

switch (expression)
     case constantValue1:
     case constantValueN: // note #1
          statementsBlock1;
          break; // optional break

     case constantValue2:
          statementsBlock2;
          break; // optional break

     case constantValue3:
          statementsBlock3;
          break; // optional break

     default:
          defaultStatementBlock;
     break; // optional break
endswitch
```

Note: (1) Multiple case statement values can be associated with a statement block

The **case** statements and the **default** statement can occur in any order in the **switch** body. The **default** statement is optional, and is activated if none of the constants in the **case** statements can be matched. The **continue** statement may be used to force the **switch** statement to re-evaluate the expression, and restart the **switch** body.

Note: (2) The compiler produces the most efficient code if the case values are a consecutive range of numbers in which case it can produce a simple linear vector table which will not only be compact, but will execute a bit faster than non-consecutive values. This does not mean that the cases need to be in this order, they are sorted during compilation to produce a linear table if possible. If the case values are not ordered, the compiler still determines the best strategy for building case statements depending on case values, eg if you use values less than 255 it can build a match table for the vectors in a linear byte array (similar to the **lookup8** function), but if the values exceed 255 it needs to build a match table with words (similar to the **lookup16** function).

```
Example:
var msg[100];
func main()
   var result;
   to(msg); putstr("This msg contains 4DGL and other words");
   print("string has ",strlen(msg), " characters\n");
   result := test1();
   print("found '4DGL' at position ", result, "\n");
repeat forever
endfunc

// Nested Switch test #1
// Uses four nested switch statements in a loop to scan character patterns.
Not a good way to scan for patterns, but a good switch test. Note, the breaks
have been commented out as they are redundant as there are no more cases in
the nested switch level.

func test1()
   var i, p;
   p := str_Ptr(msg);
   for (i := 0; i < strlen(msg) - 3; i++)
      switch (str_GetByte(p+i))
         case '4':
            switch (str_GetByte(p+i+1))
               case 'd':
               case 'D':
                  switch (str_GetByte(p+i+2))
                     case 'g':
                     case 'G':
                     switch (str_GetByte(p+i+3))
                        case 'l':
                        case 'L':
                        return i;
                     endswitch
                     //break;
                  endswitch
               //break;
            endswitch
         //break;
      endswitch
   next
   return 0;
endfunc
```

7.6.2. Switch without Expression

The second type of switch statement in the '**expressionless switch**' which allows a complete expression to be evaluated for each **case**. It can be thought of as a sequence if **if** / **endif** blocks, with the optional **break** acting as a **goto** to skip the rest of the evaluations, or **continue** to act as a **goto** to restart the switch block from the top. Overall, this type of switch usually gives a neater appearance than the **if** / **else** / **endif** construction, and is easier to visualize.

Note: (1) For this class of switch, the switch statement has no expression.

```
Syntax:

switch
     case (expression 1) // note#2
          statementsBlock1;
          break; // optional break

     case (expression 2)
          statementsBlock2;
          break; // optional break

     case (expression 3)
          statementsBlock3;
          break; // optional break
     default
          defaultStatementBlock;
     break; // optional break
endswitch
```

Note: (2) Each case must be a parenthesised comparison expression, and no semicolon follows.

```
Example:
func main()
  var c;
  gfx_Cls();
  putstr("Type keys on terminal...");
  repeat
     while((c := serin()) < 0);
        print("\n'",[CHR]c,"' is ");
           switch
               case (c < 0x20)
                   putstr(" control char");
                   break;
               case (c >= 'A' && c <= 'Z')
                   putstr(" uppercase");
               case (c >= 'a' && c <= 'z')
                   putstr(" lowercase");
               case (c == '0')
                   putstr(" zero");
                   break;
               case (lookup8(c, "13579"))
                   putstr(" odd number");
                   break;
               case (lookup8(c, "2468"))
                   putstr(" even number");
                   break;
               case (lookup8(c, "aeiouAEIOU"))
                   putstr(" vowel");
                   break;
               case(c < 'A' || c > 'z' && c < 'a' || c > 'Z')
                   putstr(" punctuation");
                   break;
               default:
                   putstr(" consonant");
           endswitch
  forever
endfunc
```

Notes:

- Note that **case** statements are evaluated in order, and the **default** clause is executed if none of the other expressions cause a **case** action.
- The **break** statement has the usual behavior and exits the switch loop, and the **continue** statement may be used to force a restart of the **switch** and re-evaluate the expressions.
- A maximum of 1000 case statements are allowed.
- The **default** statement must be last.

7.7. The Break and Continue Statements

It is possible exit from a **while, repeat, or for** loop at any time by using the *break* statement. When the execution path encounters a **break** statement the looping will stop and execution will proceed to the code immediately following the loop.

It is important to note that in the case of nested loops the **break** statement only exits the current loop leaving the outer loop to continue execution.

The **continue** statement causes all remaining code statements in a loop to be skipped and execution to be returned to the top of the loop.

```
Example:
func main()
  while (n < 20)
         n++;
         if ((n == 3) continue;
         if (n == 7) break; // output is 12456
                print(n);
  wend
:
:
endfunc
```

In the above example, **continue** statement will cause the printing of “n” be skipped when it equals 3 and the **break** statement will cause loop termination when “n” reaches 7.

8. Functions and Subroutines

8.1. func … endfunc

A function in 4DGL, just as in the C language, is a block of code that performs a specific task. Each function has a unique name and it is reusable i.e. it can be called and executed from as many different parts in a 4DGL program. A function can also optionally return a value to the calling program. Functions can also be viewed as small programs on their own.

Some of the properties of functions in 4DGL are:

- A function must have a unique name and it is this name that is used to call the function from other functions, including main().
- A function performs a specific task and the task is some distinct work that the program must perform as part of its overall operation.
- A function is an independent smaller program which can be easily removed and debugged.
- A function will always return to the calling program and can optionally return a value.

Functions have several advantages:

- Less code duplication –easier to read / update programs
- Simplifies debugging –each function can be verified separately
- Reusable code –the same functions can be used in different programs

```
Syntax:

func name([var parameter1], [var parameter2,…[var parameterN]])
   [statements]
   [return;]
   //OR
   [return (value);]
endfunc

name : Required name for the function.
var parameters : Optional list of parameters to be passed to
the function.
statements : A block of statements that make up the body of
the function.
return : Exit this function, returning control back to the
calling function (optional).
return value : Exit this function, with a variable or expression
to return a value for this function (optional).

Notes:
   - Functions must be created before they can be used.
   - Functions can optionally return a value.
```

Example1:

```
/*
This example shows how to create a function and its usage in
calling and passing parameters.
*/

func add2(var x, var y)
   var z;
   z := x + y;
   return z;
endfunc

func main()
   var a;
   a := add2(10, 4);
   print(a);
endfunc
```

Functions can now be on a single line,

Example2:

```
func myfunc(var n) print("Error: ", [STR] errorstrings[n]; endfunc
```

As long as there is no forward references that need to be calculated, the ordering of functions no longer needs to be ordered with strict backward referencing.

A 4DGL program must have a starting origin where the point of execution begins. When a 4DGL program is launched, EVE processor takes control and needs a specific starting point. This starting point is the **main()** function and every 4DGL program must have one. The main() function is the block of code that makes the whole program work.

8.2. Functions with Arguments and Return Value

In other languages and in mathematics a function is understood to be something which produces a value or a number. That is, the whole function is thought of as having a value. In 4DGL it is possible to choose whether or not a function will have a value. It is possible to make a function return a value to the place at which it was called.

```
Example:
bill = CalculateBill(data,...);
```

The variable bill is assigned to a function CalculateBill() and data are some data which are passed to the function. This statement makes it look as though CalculateBill() is a number. When this statement is executed in a program, control will be passed to the function CalculateBill() and, when it is done, this function will then hand control back. The value of the function is assigned to "bill" and the program continues. Functions which work in this way are said to return a value.

In 4DGL, returning a value is a simple matter. Consider the function CalculateBill() from the statement above:

```
func CalculateBill(var starter, var main, var dessert) //Adds up values
   var total;
   total := starter + main + dessert;
   return (total);
endfunc
```

As soon as the return statement is met CalculateBill() stops executing and assigns the value total to the function. If there were no return statement the program could not know which value it should associate with the name CalculateBill and so it would not be meaningful to speak of the function as having one value. Forgetting a return statement can ruin a program, then the value bill would just be garbage (no predictable value), presuming that the compiler allowed this to be written at all.

8.3. gosub … endsub

The **gosub** starts executing the statements at the label (subroutine name) until it reaches endsub and returns to continue after the gosub statement. The **gosub** statement can only be used within a function and all gosub labels are private within the function body. All subroutines must end with **endsub;** All labels must be followed by a colon '**:**'.

There are 2 versions of gosub as shown below:

```
Syntax1:

gosub label;
…
label:
   [statements]
endsub;

label : label, where the gosub is directing to.
```

The above version (syntax1) executes the statements at **label:.** When the **endsub;** statement is reached, execution resumes with the statement following the **gosub** statement. The code between **label:** and the **endsub;** statement is called a subroutine.

```
Example:
func myfunc()
    gosub mysub1;
    gosub mysub2;
    gosub mysub3;
    print("\nAll Done\n")
    return; // return from function *** see below

  mysub1:
     print("\nexecuted sub #1");
  endsub; // return from subroutine

  mysub2:
     print("\nexecuted sub #2");
  endsub; // return from subroutine

  mysub3:
     print("\nexecuted sub #3");
  endsub; // return from subroutine
endfunc

func main()
    myfunc();
endfunc
```

Notes:

- The gosub function can only be used within a function.
- All **gosub** labels are private within the function body.
- All subroutines must end with endsub**;**
- All **labels** must be followed by a **colon**.
- ** A common mistake is to forget the 'return' to return from a subroutine within a function.

```
Syntax2:

gosub (index), (label1, label2,.., labelN);
…
label1:
   [statements]
endsub;

label2:
   [statements]
endsub;
…
labelN:
   [statements]
endsub;
```

The above version (syntax2) uses index to **index** into the list of **gosub** labels. Execution resumes with the statement following the **gosub** statement. For example, if **index** is zero or **index** is greater than the number of labels in the list, the subroutine named by the first label in the list is executed. If **index** is one, then the second label and so on.

Note: If **index** is zero or greater than the number of labels, the first label is always executed.

Example:

```
func myfunc(var key)
    var r;
    to(COM0);      // set redirection for the next print command
                   // to the COM port
    r := lookup8(key, "fbsclx?h");
    gosub(r), (unknown,foreward,backward,set,clear,load,
    exit,help,help);
    goto done;

  help:
    putstr("Menu f,b,i,d,s,c,l or x (? or h for help)\n");
  endsub;
  unknown:
    print("\nBad command '",[CHR] key,"' ?");
    /* more code here */
  endsub;
  foreward:
    print("\nFOREWARD ");
    /* more code here */
  endsub;
  backward:
    print("\nBACKWARD ");
    /* more code here */
  endsub;
  set:
    print("\SET ");
    /* more code here */
  endsub;
  clear:
    print("\nCLEAR ");
    /* more code here */
  endsub;
  load:
    print("\nLOAD ");
    /* more code here */
  endsub;
  exit:
    print("\nEXIT");
    print("\nbye....");
     /* more code here */
    r := -1;   // signal an exit
  endsub;
  done:
    return r;
endfunc

//=============================================================
func main()
    var char;
    putstr("Open the Workshop4 terminal\n");
    putstr("Enter f,b,i,d,s,c,l or x\n");
    putstr("Enter ? or h for help\n");
    putstr("Enter x for exit\n");
    char := '?';
    goto here; // enter here to show help menu first up
    repeat
        while((char := serin()) < 0); // wait for a character
here:
        if(myfunc(char) == -1) break;
    // keep going until we get the exit command
    forever
    putstr("\nEXITING");
endfunc
//=============================================================
```

Notes:

- The indexed gosub function can only be used within a function.
- All **gosub** labels are private within the function body.
- All subroutines must end with endsub**;**
- All **labels** must be followed by a **colon**.
- If **index** is zero or greater than the number of labels, the first **label** is always executed.
- ** A common mistake is to forget the 'return' to return from a subroutine within a function.

8.4. SystemReset()

This function resets and restarts the program, It is the equivalent of a 'cold boot' (i.e. a total hardware reset). There is a 2 second delay before the program restarts, this is due to the EVE boot procedure time.

8.5. ProgramExit()

This function resets contrast to 0 and puts the display into low power sleep mode. For some devices, the only wakeup procedure is a reset or power cycle. Refer to individual module or chip specifications for information on other sleep / wakeup modes.

8.6. Argcount(function_name)

This compiler function returns then number of arguments required by a PmmC or user function. It is used often for getting the argument count when using the '@' operator when using a function pointer.

8.7. @ (the argument pointer)

Function arguments can now be passed using the special pointer operator '@'.

```
Example:
#constant rsize argcount(gfx_Rectangle)
// define rsize for number of args to gfx_Rectangle()

var rect[ rsize * 4], n;
//  an array to hold info for 4 rectangles

func main()
// initialize some default rectangle co-ords
*rect := [10,10,40,40,RED,88,10,118,40,GREEN,10,88,40,118,BLUE,
88,88,118, 118,YELLOW];

for (n:=0; n < 4*rsize ; n+=rsize  )
  gfx_Rectangle(@ rect+n);
  //draw all rectangles using arg pointer offset by n
next
repeat forever // done
endfunc

//alternatively, the ++ iterator can be employed which is a
//little more code efficient, a little faster in execution speed
//and allows a little more flexibility. Note that the iterator
//value is not 'sticky' and is reset to 1 once 'ndx++' is
//executed.
ndx:=0;
while(ndx<4)
  iterator(rsize);
  // set the iterator to the size of args for 'ndx++'

  gfx_Rectangle(@rect + ndx++);
  // draw new rectangles, bump iterator
wend
```

9. Processor Specific Internal Functions

9.1. Goldelox Internal Functions (Chip Resident)

Refer to the document: ***Goldelox Internal Functions Manual***

9.2. Picaso Internal Functions (Chip Resident)

Refer to the document: ***Picaso Internal Functions Manual***

9.3. PIXXI-28 and PIXXI-44 Internal Functions (Chip Resident)

Refer to the document: ***Pixxi Internal Functions Manual***

9.4. Diablo16 Internal Functions (Chip Resident)

Refer to the document: ***Diablo16 Internal Functions Manual***

10. Revision History

Revision	Revision Content	Revision Date
5.1	Reformatted, minor document updates	23/11/2012
5.2	Fixed Case example which has : marks in error	27/02/2013
5.3	Added some detail for DIABLO16 Processor	24/07/2013
6.0	Updated formatting and contents	01/05/2017
6.1	Reformatted, minor document updates, cosmetic changes	11/04/2019
6.2	Removed duplicate chapters and fixed formatting	21/11/2019
6.3	Adding missing information to #MODE and added #STACK section	26/08/2020

11. Legal Notice

Proprietary Information

The information contained in this document is the property of 4D Labs Semiconductors and may be the subject of patents pending or granted, and must not be copied or disclosed without prior written permission.

4D Labs Semiconductors endeavours to ensure that the information in this document is correct and fairly stated but does not accept liability for any error or omission. The development of 4D Labs Semiconductors products and services is continuous and published information may not be up to date. It is important to check the current position with 4D Labs Semiconductors. 4D Labs Semiconductors reserves the right to modify, update or makes changes to Specifications or written material without prior notice at any time.

All trademarks belong to their respective owners and are recognised and acknowledged.

Disclaimer of Warranties & Limitation of Liability

4D Labs Semiconductors makes no warranty, either expressed or implied with respect to any product, and specifically disclaims all other warranties, including, without limitation, warranties for merchantability, non-infringement and fitness for any particular purpose.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications.

Images and graphics used throughout this document are for illustrative purposes only. All images and graphics used are possible to be displayed on the 4D Labs Semiconductors range of products, however the quality may vary.

In no event shall 4D Labs Semiconductors be liable to the buyer or to any third party for any indirect, incidental, special, consequential, punitive or exemplary damages (including without limitation lost profits, lost savings, or loss of business opportunity) arising out of or relating to any product or service provided or to be provided by 4D Labs Semiconductors, or the use or inability to use the same, even if 4D Labs Semiconductors has been advised of the possibility of such damages.

4D Labs Semiconductors products are not fault tolerant nor designed, manufactured or intended for use or resale as on line control equipment in hazardous environments requiring fail – safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines or weapons systems in which the failure of the product could lead directly to death, personal injury or severe physical or environmental damage (‘High Risk Activities’). 4D Labs Semiconductors and its suppliers specifically disclaim any expressed or implied warranty of fitness for High Risk Activities.

Use of 4D Labs Semiconductors’ products and devices in 'High Risk Activities' and in any other application is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless 4D Labs Semiconductors from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any 4D Labs Semiconductors intellectual property rights.

12. Contact Information

For Technical Support: www.4dlabs.com.au/support
For Sales Support: sales@4dlabs.com.au
Website: www.4dlabs.com.au