4D LABS

4DGL

Programmers Reference Manual and Language Specifications

Document Date: 12th September 2009
Document Revision: 1.0

Table of Contents

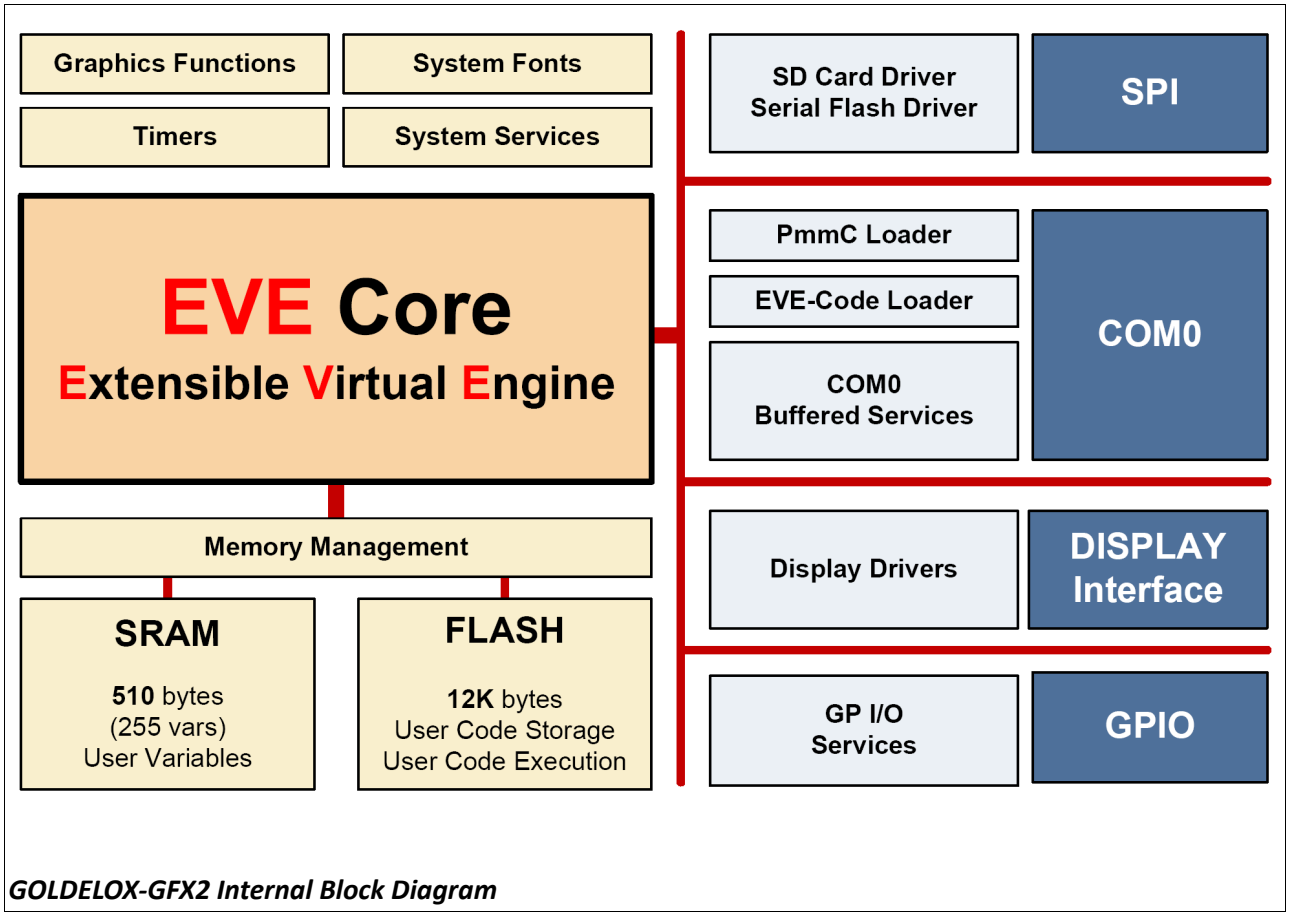
1. 4DGL INTRODUCTION

The 4D-Labs family of embedded graphics processors such as the : GOLDELOX-GFX, PICASO-GFX and the DIABLO-GFX to name a few, are powered by a highly optimised 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 optimised 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 8.



GOLDELOX-GFX2 Internal Block Diagram

2. LANGUAGE SUMMARY

The document is made up of the following sections:

- **Language Style**
  Numbers, identifiers, comments

- **Constants and Variables**
  var, #constant, #CONST...#END, #DATA...#END

- **Expressions and Operators**
  :=, &(address modifier), *, +, -, *, /, %, &(as a logial operator), |, ^, ==, !=, >, <, <=, &&, ||, !, <<, >>, ++, --

- **Language Flow Control**
  if .. else .. endif, while .. wend, repeat .. until/forever, goto

- **Functions and Subroutines**
  gosub .. endsub, func .. endfunc, return

- **Processor Specific Internal Functions (Chip Resident)**
  GOLDELOX-GFX2 Internal Functions
  PICASO-GFX Internal Functions
  DIABLO-GFX Internal Functions

3. LANGUAGE STYLE

4DGL is a case sensitive language. The colour of the text, in the 4DGL Workshop 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 instruction. Something like 'Turn Talk LED ON' is much more useful.

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.
```

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'; // buffer is 16bits per location so chars are packed
   buffer2[0] := 'EF';
   buffer2[1] := 0;
   :
   :
endfunc
```

Example:

```
var funclist[2];

//-------------------------------------------------------------------------
func foo()
   [...some code here...]
endfunc
//-------------------------------------------------------------------------
func baa()
   [...some code here...]
endfunc
//-------------------------------------------------------------------------
func main()
```

```
   funclist[0] := foo;  // load the function pointers into an array
   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.
- Local arrays are currently not supported.
- More data types will be added in future releases.

4.2 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 constants value is prepended with a **$** it becomes a complete text substitution with no validation during pre-processing.

Syntax:

```
#constant NAME value
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
```

4.3 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 delared 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

#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
```

Example:

```
#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
```

5. EXPRESSIONS and OPERATORS

Operators makes 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 performs 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 a part of even larger expressions or they can be stored in variables.

5.1 Assignment Operator

"**:=**" Assign a value to a variable:

Example:

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

5.2 Address Modifiers

"**&**" Get the address of a variable:

Example:

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

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

Example:

```
// can be used on left and/or right side of assignment operator.

a := *b;
*j := myArray;
*d := *s;
```

5.3 Arithmetic Operators

"**+**" Addition:

Example:

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

"-" Subtraction:

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

"*" Multiplication:

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

GOLDELOX-GFX2: the overflow (bits 16 to 31) is placed into the **SYS_OVERFLOW** register which can be read by the **OVF()** function.

"/" Division:

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

GOLDELOX-GFX2: the remainder is placed into the **SYS_OVERFLOW** register which can be read by the **OVF()** function.

"%" 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 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.

5.4 Logical Operators

Performs the bit wise operation and stores the value in to the assigned variable.

"&" Perform a bitwise AND operation:

Example:
```
i := k & j;
```

"|" Perform a bitwise OR operation:

Example:
```
i := k | j;
```

"**&**" Perform a bitwise XOR operation:

Example:
```
i := k ^ j;
```

5.5 Comparison Operators

These comparisons are most frequently used to control an if statement or a repeat or a while loop. These will be introduced in a later chapter. Note that == is used in comparisons and := is used in assignments.

"**==**" Equals test:

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

"**!=**" Not Equals test:

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

"**>**" Greater Than test:

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

"**>=**" Greater Than or Equals test:

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

"**<**" Less Than test:

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

"**<=**" Less Than or Equals test:

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

"**&&**" Logical AND test:

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

"**||**" Logical OR test:

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

5.6 Bit Shifting

"**<<**" Shift Left:

Example:
```
i := k << j;
```

Return the result of shifting the bits of k to the left j bits and store it in i. When left shifting occurs, the most significant bit goes out while a zero moves in as a least significant bit for every single bit shift.

GOLDELOX-GFX2: the most significant bit goes out and into the **SYS_OVERFLOW** register which can be read by the **OVF()** function.

"**>>**" Shift Right:

Example:
```
var myvar, temp, MSB;
…
temp := myvar & 0xff00;
MSB := myvar >> 8;
```

Return the result of shifting the bits of myvar to the right by 8 bits and store it in MSB. When Right Shifting occurs, the least significant bit goes out while a zero moves in as a most significant bit for every single bit shift.

GOLDELOX-GFX2: the least significant bit goes out and into the **SYS_OVERFLOW** register which can be read by the **OVF()** function.

5.7 Short Hand Notations

"**var++**" Post-Increment:

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

"**++var**" Pre-Increment:

Example:
```
x := ++b*a;
```

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

"**var--**" Post-Decrement:

Example:

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

"**--var**" Pre-Decrement:

Example:

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

6. LANGUAGE FLOW CONTROL

6.1 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
```

6.2 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.

```
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, 20, rad++, color++);
         gfx_Ellipse(120, 160, rad++, 20, color++);
         gfx_Line(120, 160, 20, rad++, color++);
         gfx_Rectangle(10, 10, rad++, rad++, color++);
         display_Vsync(); // wait for VSYNC as a delay
      wend
   wend
endfunc
```

6.3 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**.

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 the 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);
```

6.4 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. FUNCTIONS and SUBROUTINES

7.1 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.
```

```
Example:

/*
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
```

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.

7.2 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.

7.3 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;
```

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 Workshop 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. PROCESSOR SPECIFIC INTERNAL FUNCTIONS

8.1 GOLDELOX-GFX2 Internal Functions (Chip Resident)

Refer to the external document:
GOLDELOX-GFX2-4DGL-Internal-Functions.pdf

8.2 PICASO-GFX Internal Functions (Chip Resident)

Refer to the external document:
PICASO-GFX-4DGL-Internal-Functions.pdf

8.3 DIABLO-GFX Internal Functions (Chip Resident)

Refer to the external document:
DIABLO-GFX-4DGL-Internal-Functions.pdf

Proprietary Information

The information contained in this document is the property of 4D Labs Pty. Ltd. and may be the subject of patents pending or granted, and must not be copied or disclosed with out prior written permission.

4D Labs 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 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.

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

Disclaimer of Warranties & Limitation of Liability

4D Labs makes no warranty, either express 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.

In no event shall 4D Labs 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, or the use or inability to use the same, even if 4D Labs has been advised of the possibility of such damages.

Use of 4D Labs' devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless 4D Labs from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any 4D Labs intellectual property rights.