4D Labs

# GOLDELOX-PoGa
# 4DGL Internal Functions

Document Date: 18th November 2011
Document Revision: 2.0

## Table of Contents

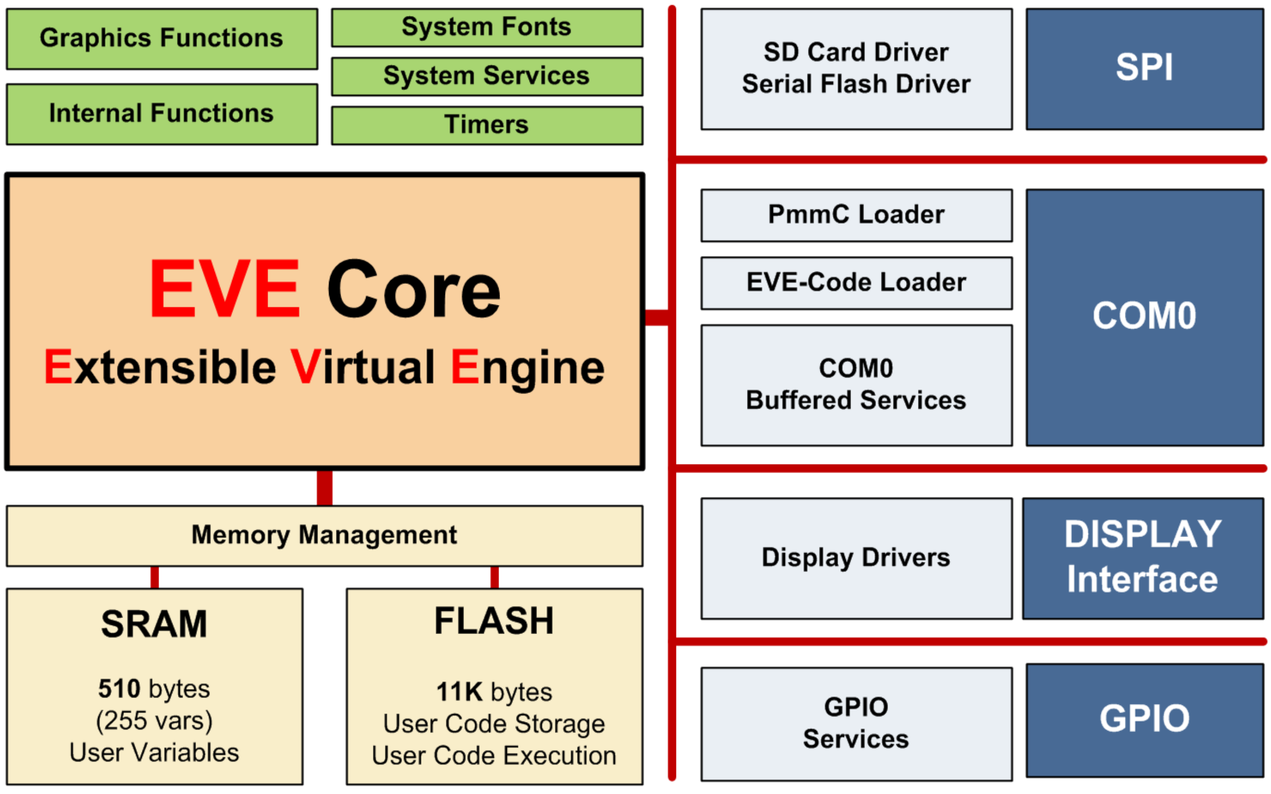
# 1. 4DGL Introduction

The 4D-Labs family of embedded graphics processors such as the : GOLDELOX-PoGa, GOLDELOX-GFX2, 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 internal (chip-resident) functions available for the GOLDELOX-PoGa. This document should be used in conjunction with “4DGL-Programmers-Reference-Manual” document.



***GOLDELOX-PoGa Internal Block Diagram***

# 2. GOLDELOX-PoGa Chip-Resident Functions Summary

The following is a summary of chip-resident 4DGL functions within the GOLDELOX-PoGa graphics controller. The document is made up of the following sections:

**2.1 Joystick Functions:**

- joystick()
- Joyval()

**2.2 Memory Access Functions:**

- peekB(address)
- peekW(address)
- pokeB(address, byte_value)
- pokeW(address, word_value)

**2.3 User Stack Functions:**

- setsp(index)
- getsp()
- pop()
- push(value)
- drop(n)
- call()
- exec(functionPtr, argCount)

**2.4 Maths Functions:**

- ABS(value)
- MIN(value1, value2)
- MAX(value1, value2)
- SWAP(&var1, &var2)
- SIN(angle)
- COS(angle)
- RAND()
- SEED(number)
- SQRT(number)
- OVF ()

**2.5 Text and String Functions:**

- txt_MoveCursor(line, column)
- putch(char)
- putstr(pointer)
- putnum(format, value)
- print(...)
- to(outstream)
- charwidth('char')
- charheight('char')
- strwidth(pointer)
- strheight()
- strlen(pointer)

- txt_Set(function, value)
  **txt_Set shortcuts:**
  - txt_FGcolour(colour)
  - txt_BGcolour(colour)
  - txt_FontID(id)
  - txt_Width(multiplier)
  - txt_Height(multiplier)
  - txt_Xgap(pixelcount)
  - txt_Ygap(pixelcount)
  - txt_Delay(millisecs)
  - txt_Opacity(mode)
  - txt_Bold(mode)
  - txt_Italic(mode)
  - txt_Inverse(mode)
  - txt_Underlined(mode)
  - txt_Attributes(value)

**2.6 Graphics Functions:**

- gfx_Cls()
- gfx_ChangeColour(oldColour, newColour)
- gfx_Circle(x, y, radius, colour)
- gfx_CircleFilled(x, y, radius, colour)
- gfx_Line(x1, y1, x2, y2, colour)
- gfx_Hline(y, x1, x2, colour)
- gfx_Vline(x, y1, y2, colour)
- gfx_Rectangle(x1, y1, x2, y2, colour)
- gfx_RectangleFilled(x1, y1, x2, y2, colour)
- gfx_Polyline(n, vx, vy, colour)
- gfx_Polygon(n, vx, vy, colour)
- gfx_Triangle(x1, y1, x2, y2, x3, y3, colour)
- gfx_Dot()
- gfx_Bullet(radius)
- gfx_OrbitInit(&x_dest, &y_dest)
- gfx_Orbit(angle, distance)
- gfx_PutPixel(x, y, colour)
- gfx_GetPixel(x, y)
- gfx_MoveTo(xpos, ypos)
- gfx_MoveRel(xoffset, yoffset)
- gfx_LineTo(xpos, ypos)
- gfx_LineRel(xpos, ypos)
- gfx_BoxTo(x2, y2)
- gfx_SetClipRegion()
- gfx_ClipWindow(x1, y1, x2, y2)
- gfx_FocusWindow()
- gfx_SpriteSet(bitmaps, colors, palette)
- gfx_BlitSprite(spritenumber, palette, xpos, ypos, orientation)

- rect_Intersect(&rect1, &rect2)
- rect_Within(&rect1, &rect2)
- gfx_Set(function, value)

  **gfx_Set shortcuts:**
  - gfx_PenSize(mode)
  - gfx_BGcolour(colour)
  - gfx_ObjectColour(colour)
  - gfx_Clipping(mode)
  - gfx_TransparentColour(colour)
  - gfx_Transparency(mode)
  - gfx_FrameDelay(delay)
  - gfx_ScreenMode(mode)
  - gfx_OutlineColour(colour)
  - gfx_Contrast(value)
  - gfx_LinePattern(pattern)
  - gfx_ColourMode(mode)

### 2.7 Display I/O Functions:

- disp_setGRAM(x1, y1, x2, y2)
- disp_WriteControl(value)
- disp_WriteByte(value)
- disp_WrGRAM(value)
- disp_RdGRAM()
- disp_ReadWord()
- disp_BlitPixelFill(colour, count)
- disp_BlitPixelsToMedia()
- disp_BlitPixelsFromMedia(pixelcount)
- disp_SkipPixelsFromMedia(pixelcount)
- disp_BlitPixelsToArray(dest, count)
- disp_BlitPixelsFromArray(source, count)
- disp_Scroll(x1, y1, x2, y2, mode, lines, bufptr)

### 2.8 Media Functions (SD/SDHC memory Card or Serial Flash chip):

- media_Init()
- media_SetAdd(HIword, LOword)
- media_SetSector(HIword, LOword)
- media_ReadByte()
- media_ReadWord()
- media_WriteByte(byte_val)
- media_WriteWord(word_val)
- media_Flush()
- media_Image(x, y)
- media_Video(x, y)
- media_VideoFrame(x, y, frameNumber)
- media_SelectGCIimage(entrynum, frame,mode)
- media_Offset(sector)
- media_LoadArray(dest, count)

- media_StoreArray(source, count)
- media_LoadImageHeader()
- media_SetScanLine(line, offset)
- media_PoGaFile(filenumber)

### 2.9 PoGa File system operations:

- RunProgram(page)
- LoadProgram(ByteCount, page)

### 2.10 Extended Functions:

- func iterator(offset)
- EVE_SP()

### 2.11 Serial (UART) Communications Functions:

- serin()
- serout(char)
- setbaud(rate)
- serout(char)
- setbaud(rate)
- com_Reset()
- com_Count()
- com_Full()
- com_Error()
- com_Sync()
- com_TX(buf, bufsize)
- com_TX_Count()
- com_CSUM_8(buf,count)
- com_CRC_16(buf, count)
- com_CRC_MODBUS(buf, count)
- com_CRC_CCITT(buf, count, seed)
- sys_EventsPostpone()
- sys_EventsResume()

### 2.12 Sound and Tune (RTTTL) Functions:

- beep(note, duration)
- tune_Play(tuneptr)
- tune_Pause()
- tune_Continue()
- tune_Stop()
- tune_End()
- tune_Playing()

### 2.13 General Purpose Functions:

- pause(time)
- lookup8 (key, byteConstList )
- lookup16 (key, wordConstList )

## 2.1 Joystick Functions

**Summary of Functions in this section:**

- joystick()
- Joyval()

### 2.1.1 joystick()

| | | |
|---|---|---|
| **Syntax** | **joystick();** | |
| **Arguments** | **none** | |
| **Returns** | **value** | |
| | **value** | Returns the joystick value. |

**Description**

Returns the value of the Joystick position (6 position switch implementation).

The JOYSTICK values are:

| **Value** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **Status** | Released | UP | LEFT | DOWN | RIGHT | BTNB | BTNA |

**Note:** The joystick input uses IO1 utilizing the A/D converter. Each switch is connected to junction of 2 resistors that form a  unique voltage divider circuit.

**Example**

```
joy := joystick();                // read the joystick
if (joy == 0) putstr("     ");
if (joy == 1) putstr(" UP");
if (joy == 2) putstr("LEFT");
if (joy == 3) putstr("DOWN");
if (joy == 4) putstr("RIGHT");
if (joy == 5) putstr("BTNB");
if (joy == 6) putstr("BTNA");
```

### 2.1.2 joyval()

| | | |
|---|---|---|
| **Syntax** | **joyval();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **result** | |
| | **result** | Values read should be approximately:-<br>Value = 255 : RELEASED<br>Value = 211 : UP<br>Value = 174 : LEFT<br>Value = 128 : DOWN<br>Value = 81 : RIGHT<br>Value = 52 : BTNB<br>Value = 27 : BTNA |
| | | |
| **Description** | Read the raw A/D joystick value. Note that combinations of buttons return intermediate values these 'raw' values may be able to be put to use for multiple button press detection. | |
| | | |
| **Example** | `var := Joyval();` | |

## 2.2 Memory Access Functions

**Summary of Functions in this section:**

- peekB(address)
- peekW(address)
- pokeB(address, byte_value)
- pokeW(address, word_value)

### 2.2.1 peekB(address)

| | | |
|---|---|---|
| **Syntax** | **peekB(address);** | |
| | | |
| **Arguments** | **address** | |
| | **address** | The address of a memory byte. The address is usually a pre-defined system register address constant, (see the address constants for all the system byte sized registers in section 3, table 3.1). |
| | The argument can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **byte_value** | |
| | **byte_value** | The 8 bit value stored at **address.** |
| | | |
| **Description** | This function returns the 8 bit value that is stored at **address.**<br>**Note:** the peekB(..) and pokeB(..) functions are usually only used with internal system byte registers using the pre-defined constants. If peekB(..) or pokeB(..) are used to access other locations, the address must be doubled to get the correct pointer address. | |
| | | |
| **Example** | `var myvar;`<br>`myvar := peekB(GFX_XMAX) + 1;`<br>This example places the width of the display (horizontal resolution in pixel units) in **myvar**. | |

## 2.2.2 peekW(address)

| | | |
|---|---|---|
| **Syntax** | **peekW(address);** | |
| | | |
| **Arguments** | **address** | |
| | **address** | The address of a memory word. The address is usually a pre-defined system register address constant, (see the address constants for all the system word sized registers in section 3, table 3.2). |
| | The argument can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **word_value** | |
| | **word_value** | The 16 bit value stored at **address.** |
| | | |
| **Description** | This function returns the 16 bit value that is stored at **address.** | |
| | | |
| **Example** | `var myvar;`<br>`myvar := peekW(SYSTEM_TIMER_LO);`<br><br>This example places the low word of the 32 bit system timer in **myvar**.<br>The equivalent operation using a pointer is:-<br>myvar := *TIMER2; | |

### 2.2.3 pokeB(address, byte_value)

| | | |
|---|---|---|
| **Syntax** | **pokeB(address, byte_value);** | |
| | | |
| **Arguments** | **address, byte_value** | |
| | **address** | The address of a memory byte. The address is usually a pre-defined system register address constant, (see the address constants for all the system byte sized registers in section 3, table 3.1). |
| | **byte_value** | The lower 8 bits of **byte_value** will be stored at **address.** |
| | The arguments can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **boolean** | |
| | **boolean** | Returns **TRUE** if poke address was a legal address (usually ignored). |
| | | |
| **Description** | This function writes a 8 bit value to a location specified by **address.**<br>**Note:** the peekB(..) and pokeB(..) functions are usually only used with internal system byte registers using the pre-defined constants. If peekB(..) or pokeB(..) are used to access other locations, the address must be doubled to get the correct pointer address. | |
| | | |
| **Example** | `pokeB(CLIP_TOP, 10);`<br>This example manually adjusts the top clipping point to 10 pixels down from top of screen. | |

### 2.2.4 pokeW(address, word_value)

| | | |
|---|---|---|
| **Syntax** | **pokeW(address, word_value);** | |
| | | |
| **Arguments** | **address, word_value** | |
| | **address** | The address of a memory word. The address is usually a pre-defined system register address constant, (see the address constants for all the system word sized registers in section 3, table 3.2). |
| | **word_value** | The 16 bit word_value will be stored at **address.** |
| | The arguments can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **boolean** | |
| | **boolean** | Returns **TRUE** if poke address was a legal address (usually ignored). |
| | | |
| **Description** | This function writes a 16 bit value to a location specified by **address.** | |
| | | |
| **Example** | `pokeW(TIMER2, 5000);`<br><br>This example sets TIMER2 to 5 seconds.<br>The equivalent operation using a pointer is:<br>`*TIMER2 := 5000;` | |

## 2.3 User Stack Functions

EVE provides all the requirement for a user stack to aid in development of stack based processing e.g. for interpreters and fast raster drawings. The stack is at a fixed location (it is at the base of the user memory) . The stack pointer always expects the stack to be here – it is hard micro-coded internally.

If none of the stack functions are used, the stack can be disregarded as it will not influence any other program dynamics – the memory can be used for other purposes. If a user stack is required, it must be configured as the first array in the users program. The stack pointer always points to the current item on top of the stack.

**Note: If the stack pointer is zero, there are no items on the stack.**

**Typically, your program will look like this:**

```
// the user stack MUST be the first storage in you program
var mystack[20];          // A 20 word stack. The stack must be the first array in the program.
var myvar1, myvar2;       // etc
```

**Summary of Functions in this section:**

- setsp(index)
- getsp()
- pop()
- push(value)
- drop(n)
- call()
- exec(functionPtr, argCount)

## 2.3.1 setsp(index)

| | | |
|---|---|---|
| **Syntax** | **setsp(index);** | |
| | | |
| **Arguments** | **index** | |
| | **index** | This argument is used to set the users SP to the required position. The stack pointer is set to zero during power-up initialisation. |
| | The argument can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | The users stack pointer is zeroed at power up, but it is sometimes necessary to alter the stack pointer  for various reasons, such as running multiple concurrent stacks, or resetting to a known position as part of an error recovery process. | |
| | | |
| **Example** | `setsp(0);    // reset the stack pointer`<br>This example sets the users stack pointer to 'empty' | |

## 2.3.2 getsp()

| | | |
|---|---|---|
| **Syntax** | **getsp();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **index** | |
| | **index** | The current stack index. |
| | | |
| **Description** | This function returns the current stack index into the stack array. If the index is zero, there are no items on the stack. | |
| | | |
| **Example** | `push(1234);`<br>`print(getsp());    // print the stack index`<br><br>This example will print '1234' assuming there are no other items on the stack. | |

### 2.3.3 pop()

| | | |
|---|---|---|
| **Syntax** | **pop();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **value** | |
| | **value** | The value at current stack pointer index. |
| | | |
| **Description** | This function returns the value at the current stack pointer index. The stack pointer is then decremented, so it now points to the item below. If the stack pointer is zero, (ie a pop was performed on an empty stack) the function returns 0 and the stack pointer is not altered (ie it remains at 0). | |
| | | |
| **Example** | `push(100);`<br>`push(200);`<br>`print(pop()+ pop());`<br>This example prints '300' and the stack pointer is reduced by 2 | |

### 2.3.4 push(value)

| | | |
|---|---|---|
| **Syntax** | **push(value);** | |
| | | |
| **Arguments** | **value** | |
| | **value** | Argument to be pushed to the user stack. |
| | The argument can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Increment the user stack pointer first and then places the item into the user stack array at the current position. The stack pointer is now pointing to this new item. | |
| | | |
| **Example** | `Myvar := 10;`<br>`push(1234);`<br>`push(5678);`<br>`push(myvar);`<br><br>This example pushes 3 items to the user stack | |

### 2.3.5 drop(n)

| **Syntax** | **drop(n);** | |
|---|---|---|
| | | |
| **Arguments** | **n** | |
| | **n** | Specifies the number of items to be dropped from the stack. |
| | The argument can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Decrements the user stack pointer determined by the value n. If **n** exceeds the stack index, the stack pointer is zeroed. | |
| | | |
| **Example** | `myvar := 10;`<br>`push(1234);`<br>`push(5678);`<br>`push(myvar);`<br>`drop(2);`<br>This example decrements the stack pointer by 2, effectively dropping '**myvar**' and '**5678**' from the stack, the next pop would yield **1234**. | |

### 2.3.6 call()

| **Syntax** | **call();** | |
|---|---|---|
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **value** | |
| | **value** | If the called function returns a value then it is available. |
| | | |
| **Description** | Calls the specified function, the arguments to the called function are from the stack. The stacked parameters are consumed and the stack pointer is altered to match the number of arguments that were consumed. | |
| | | |
| **Example** | `push(10);`<br>`push(10);`<br>`push(50);`<br>`push(50);`<br>`push(0xFFFF);`<br>`push(gfx_RectangleFilled); // push the function call address`<br>`push(5);                   // push the argument count`<br><br>`//~~~~~~~`<br><br>`call();`<br><br>This example takes the function argument count, function pointer, and argument pointer from the top of the stack and calls the function using the stacked parameters. The 7 arguments on the stack are discarded. | |

## 2.3.7 exec(functionPtr, argCount)

| | | |
|---|---|---|
| **Syntax** | **exec(functionPtr, argCount);** | |
| | | |
| **Arguments** | **functionPtr, argCount** | |
| | **functionPtr** | A pointer to a function which will utilise the stacked arguments. |
| | **argCount** | The count of arguments on the stack that are to be passed to the function call. |
| | The arguments can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **value** | |
| | value | If the called function returns a value then it is available. |
| | | |
| **Description** | Calls the specified function, passing the arguments to the called function from the stack. The stack and stack pointer are not altered. | |

**Example**

```
Push(50);      // set some arbitrary values on the stack
push(50);
push(10);
push(YELLOW);

//~~~~~~~

exec(gfx_Circle,4);  // exec the circle function using
                     // the stacked parameters
```

This example draws a circle using the stacked parameters. The stacked parameters and the stack pointer are not altered.

## 2.4 Maths Functions

**Summary of Functions in this section:**

- ABS(value)
- MIN(value1, value2)
- MAX(value1, value2)
- SWAP(&var1, &var2)
- SIN(angle)
- COS(angle)
- RAND()
- SEED(number)
- SQRT(number)
- OVF ()

### 2.4.1 ABS(value)

| **Syntax** | **ABS(value);** | |
|---|---|---|
| | | |
| **Arguments** | **value** | |
| | **value** | a variable, array element, expression or constant. |
| | The argument can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **value** | |
| | **value** | Returns the absolute value. |
| | | |
| **Description** | This function returns the absolute value of **value.** | |
| | | |
| **Example** | `var myvar, number;`<br>`number := -100;`<br>`myvar := ABS(number * 5);`<br>This example returns 500 in variable **myvar**. | |

### 2.4.2 MIN(value1, value2)

| **Syntax** | **MIN(value1, value2);** | |
|---|---|---|
| | | |
| **Arguments** | **value1, value2** | |
| | **value1** | a variable, array element, expression or constant. |
| | **value2** | a variable, array element, expression or constant. |
| | The arguments can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **value** | |
| | **value** | the smaller of the two  values. |
| | | |
| **Description** | This function returns the the smaller of **value1** and **value2**. | |
| | | |
| **Example** | `var myvar, number1, number2;`<br>`number1 := 33;`<br>`number2 := 66;`<br>`myvar := MIN(number1, number2);`<br><br>This example returns 33 in variable **myvar**. | |

### 2.4.3 MAX(value1, value2)

| | | |
|---|---|---|
| **Syntax** | **MAX(value1, value2);** | |
| | | |
| **Arguments** | **value1, value2** | |
| | **value1** | a variable, array element, expression or constant. |
| | **value2** | a variable, array element, expression or constant. |
| | The arguments can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **value** | |
| | **value** | the larger of the two  values. |
| | | |
| **Description** | This function returns the the larger of **value1** and **value2**. | |
| | | |
| **Example** | `var myvar, number1, number2;`<br>`number1 := 33;`<br>`number2 := 66;`<br>`myvar := MAX(number1, number2);`<br><br>This example returns 66 in variable **myvar**. | |

### 2.4.4 SWAP(&var1, &var2)

| | | |
|---|---|---|
| **Syntax** | **SWAP(&var1, &var2);** | |
| | | |
| **Arguments** | **&var1, &var2** | |
| | **&var1** | The address of the first variable. |
| | **&var2** | The address of the second variable. |
| | The arguments can only be a variable or an array element. | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Given the addresses of two variables (var1 and var2), the values at these addresses are swapped. | |
| | | |
| **Example** | `var number1, number2;`<br>`number1 := 33;`<br>`number2 := 66;`<br>`SWAP(&number1, &number2);`<br>This example swaps the values in **number1** and **number2**. After the function is executed, **number1** will hold 66, and **number2** will hold 33. | |

### 2.4.5 SIN(angle)

| | | |
|---|---|---|
| **Syntax** | **SIN(angle);** | |
| | | |
| **Arguments** | **angle** | |
| | **angle** | The angle  in degrees. (Note: The input value is automatically shifted to lie within 0-359 degrees) |
| | The argument can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **result** | |
| | **result** | The sine in radians of an argument specified in degrees. The returned value range is from 127 to -127 which is a more useful representation for graphics work. The real sine values vary from 1.0 to -1.0 so appropriate scaling must be done in user code as required. |
| | | |
| **Description** | This function returns the sine of an **angle** | |
| | | |
| **Example** | `var myvar, angle;`<br>`angle := 133;`<br>`myvar := SIN(angle);`<br><br>This example returns 92 in variable **myvar**. | |

## 2.4.6 COS(angle)

| | | |
|---|---|---|
| **Syntax** | **COS(angle);** | |
| | | |
| **Arguments** | **angle** | |
| | **angle** | The angle  in degrees. (Note: The input value is automatically shifted to lie within 0-359 degrees) |
| | The argument can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **result** | |
| | **result** | The cosine in radians of an argument specified in degrees. The returned value range is from 127 to -127 which is a more useful representation for graphics work. The real sine values vary from 1.0 to -1.0 so appropriate scaling must be done in user code as required. |
| | | |
| **Description** | This function returns the cosine of an **angle** | |
| | | |
| **Example** | `var myvar, angle;`<br>`angle := 133;`<br>`myvar := COS(angle);`<br>This example returns -86 in variable **myvar**. | |

## 2.4.7 RAND()

| | | |
|---|---|---|
| **Syntax** | **RAND();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **value** | |
| | **value** | Returns a pseudo random signed number ranging from -32768 to +32767 each time the function is called. The random number generator may first be seeded by using the SEED(number) function. The seed will generate a pseudo random sequence that is repeatable. You can use the modulo operator (%) to return a number within a certain range, eg n := RAND() % 100; will return a random number between -99 and +99. If you are using random number generation for random graphics points, or only require a positive number set, you will need to use the ABS function so only a positive number is returned, eg: X1 := ABS(RAND() % 100); will set co-ordinate X1 between 0 and 99. Note that if the random number generator is not seeded, the first number returned after reset or power up will be zero. This is normal behavior. |
| | | |
| **Description** | This function returns a pseudo random signed number ranging from -32768 to +32767 | |
| | | |
| **Example** | `SEED(1234);`<br>`print(RAND(),", ",RAND());`<br><br>This example will print<br>**`3558, 1960`**<br>to the display. | |

### 2.4.8 SEED(number)

| | | |
|---|---|---|
| **Syntax** | **SEED(number);** | |
| | | |
| **Arguments** | **number** | |
| | **number** | Specifies the seed value for the pseudo random number generator. |
| | The argument can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | This function seeds the pseudo random number generator so it will generate a new repeatable sequence. The seed value can be a positive or negative number. | |
| | | |
| **Example** | `SEED(-50);`<br>`print(RAND(),", ",RAND());`<br><br>This example will print<br>**`30129, 27266`**<br>to the display. | |

## 2.4.9 SQRT(number)

| | | |
|---|---|---|
| **Syntax** | **SQRT(number);** | |
| | | |
| **Arguments** | **number** | |
| | **number** | Specifies the positive number for the SQRT function. |
| | The argument can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **value** | |
| | **value** | This function returns the **integer square root** which is the greatest integer less than or equal to the square root of **number**. |
| | | |
| **Description** | This function returns the **integer square root** of a number. | |
| | | |
| **Example** | `var myvar;`<br>`myvar := SQRT(26000);`<br><br>This example returns 161 in variable **myvar** which is the **integer square root** of 26000. | |

## 2.4.10 OVF()

| | | |
|---|---|---|
| **Syntax** | **OVF();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **value** | |
| | **value** | the high order 16 bits from certain math and shift functions. |
| | | |
| **Description** | This function returns the high order 16 bits from certain math and shift functions. It is extremely useful for calculating 32 bit address offsets for MEDIA access.<br>It can be used with the shift operations, addition, subtraction, multiplication and modulus operations. | |
| | | |
| **Example** | `var loWord, hiWord;`<br>`loWord := 0x2710 * 0x2710;     // (10000 * 10000 in hex format)`<br>`hiWord := OVF();`<br>`print ("0x", [HEX] hiWord,  [HEX] loWord);`<br><br>This example will print<br>**`0x05F5E100`**<br>to the display , which is 100,000,000 in hexadecimal | |

## 2.5 Text and String Functions

**Summary of Functions in this section:**

- txt_MoveCursor(line, column)
- putch(char)
- putstr(pointer)
- putnum(format, value)
- print(...)
- to(outstream)
- charwidth('char')
- charheight('char')
- strwidth(pointer)
- strheight()
- strlen(pointer)
- txt_Set(function, value)

  **txt_Set shortcuts:**

  - txt_FGcolour(colour)
  - txt_BGcolour(colour)
  - txt_FontID(id)
  - txt_Width(multiplier)
  - txt_Height(multiplier)
  - txt_Xgap(pixelcount)
  - txt_Ygap(pixelcount)
  - txt_Delay(millisecs)
  - txt_Opacity(mode)
  - txt_Bold(mode)
  - txt_Italic(mode)
  - txt_Inverse(mode)
  - txt_Underlined(mode)
  - txt_Attributes(value)

### 2.5.1 txt_MoveCursor(line, column)

| | | |
|---|---|---|
| **Syntax** | **txt_MoveCursor(line, column);** | |
| | | |
| **Arguments** | **line, column** | |
| | **line** | Holds a positive value for the required line position. |
| | **newColour** | Holds a positive value for the required column position. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Moves the origin to a screen position set by line and column parameters. The line and column position is calculated, based on the size and scaling factor for the currently selected font. When text is outputted to screen it will be displayed from this position. The text position could also be set with gfx_MoveTo(...); if required to set the text position to an exact pixel location. Note that lines and columns start from 0, so line 0 , column 0 is the top left corner of the display. | |
| | | |
| **Example** | `txt_MoveCursor(4, 9);`<br>This example moves the text origin to the 5th line and the 10th column. | |

### 2.5.2 putch(char)

| | | |
|---|---|---|
| **Syntax** | **putch(char);** | |
| | | |
| **Arguments** | **char** | |
| | **char** | Holds a positive value for the required character. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | **putch** prints single characters to the current output stream, usually the display. | |
| | | |
| **Example** | `var v;`<br>`v := 0x39;`<br>`putch(v);    // print the number 9 to the current display location`<br>`putch('\n'); // newline` | |

### 2.5.3 putstr(pointer)

| | | |
|---|---|---|
| **Syntax** | **putstr(pointer);** | |
| | | |
| **Arguments** | **pointer** | |
| | **pointer** | A string constant or pointer to a string. |
| | The argument can be a string constant or pointer to a string, a pointer to an array, or a pointer to a data statement. | |
| | | |
| **Returns** | **source** | |
| | **source** | Returns the pointer to the item that was printed. |
| | | |
| **Description** | **putstr** prints a string to the current output stream, usually the display. The argument can be a string constant, a pointer to a string, a pointer to an array, or a pointer to a data statement.<br>**Note:** **putstr** is more efficient that **print** for printing single strings.<br>The output of **putstr** can be redirected to the communications port, the media, or memory using the **to(...);** function.<br>A string constant is automatically terminated with a zero.<br>A string in a data statement is not automatically terminated with a zero.<br>All variables in 4DGL are 16bit, if an array is used for holding 8 bit characters, each array element packs 1 or 2 characters. | |
| | | |

**Example**

```
//==================================================
// Example #1 - print a string constant
//==================================================

putstr("HELLO\n"); //simply print a string constant at current origin


//==================================================
// Example #2 - print string via pointer
//==================================================
var p;                      // a var for use as a pointer
p := "String Constant\n";   // assign a string constant to pointer s
putstr(p);                  // print the string using the pointer
putstr(p+8);                // print, offsetting into the string


//==================================================
// Example #3 - printing strings from data table
//==================================================

#DATA
    byte message "Week",0
    word days sun,mon,tue,wed,thu,fri,sat // pointers to data items
    byte sun "Sunday\n\0"
    byte mon "Monday\n\0"
    byte tue "Tuesday\n\0"
    byte wed "Wednesday\n\0"
    byte thu "Thursday\n\0"
    byte fri "Friday\n\0"
    byte sat "Saturday\n\0"
#END

var n;
putstr
n:=0;
while(n < 7)
    putstr(days[n++]);  // print the days
wend
```

### 2.5.4 putnum(format, value)

| **Syntax** | **putnum(format, value);** | |
|---|---|---|
| | | |
| **Arguments** | **format, value** | |
| | **format** | A constant that specifies the number format. |
| | **value** | The number to be printed. |

**Number formatting bits supplied by format**

```
bit 15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0
    |  |  |  |   \___ ___/  \__ __/  \_____ _____/
    |  |  |  |       V         V           V
    |  |  |  |       |         |           |
    |  |  |  |       |         |           |
    |  |  |  |  (nb 0 = 16)    |           |____BASE (usually 2,10 or 16)
    |  |  |  |   digit count   |
    |  |  |  |                 |___reserved (not used on GOLDELOX)
    |  |  |  |
    |  |  |  |
    |  |  |  |
    |  |  |  |
    |  |  |  |
    |  |  |  |______ 1 = leading zeros included
    |  |  |          0 = leading zeros suppressed
    |  |  |
    |  |  |
    |  |  |_______ 1 = leading zero blanking
    |  |
    |  |_____ sign bit (0 = signed, 1 = unsigned)
    |
    |______ 1 = space before unsigned number
```

**Pre-Defined format constants quick reference**

| DECIMAL | | | UNSIGNED DECIMAL | | | HEX | | | BINARY | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| DEC | DECZ | DECZB | UDEC | UDECZ | UDECZB | HEX | HEXZ | HEXZB | BIN | BINZ | BINZB |
| DEC1 | DEC1Z | DEC1ZB | UDEC1 | UDEC1Z | UDEC1ZB | HEX1 | HEX1Z | HEX1ZB | BIN1 | BIN1Z | BIN1ZB |
| DEC2 | DEC2Z | DEC2ZB | UDEC2 | UDEC2Z | UDEC2ZB | HEX2 | HEX2Z | HEX1ZB | BIN2 | BIN2Z | BIN2ZB |
| DEC3 | DEC3Z | DEC3ZB | UDEC3 | UDEC3Z | UDEC3ZB | HEX3 | HEX3Z | HEX1ZB | BIN3 | BIN3Z | BIN3ZB |
| DEC4 | DEC4Z | DEC4ZB | UDEC4 | UDEC4Z | UDEC4ZB | HEX4 | HEX4Z | HEX1ZB | BIN4 | BIN4Z | BIN4ZB |
| DEC5 | DEC5Z | DEC5ZB | UDEC5 | UDEC5Z | UDEC5ZB | | | | BIN5 | BIN5Z | BIN5ZB |
| | | | | | | | | | BIN6 | BIN6Z | BIN6ZB |
| | | | | | | | | | BIN7 | BIN7Z | BIN7ZB |
| | | | | | | | | | BIN8 | BIN8Z | BIN8ZB |
| | | | | | | | | | BIN9 | BIN9Z | BIN9ZB |
| | | | | | | | | | BIN10 | BIN10Z | BIN10ZB |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | BIN11 | BIN11Z | BIN11ZB |
| | | | | | | | | | BIN12 | BIN12Z | BIN12ZB |
| | | | | | | | | | BIN13 | BIN13Z | BIN13ZB |
| | | | | | | | | | BIN14 | BIN14Z | BIN14ZB |
| | | | | | | | | | BIN15 | BIN15Z | BIN15ZB |
| | | | | | | | | | BIN16 | BIN16Z | BIN16ZB |

| **Returns** | **field** | |
|---|---|---|
| | **field** | Returns the the default width of the numeric field (digit count), usually ignored. |

**Description**

**putnum** prints a 16bit number in various formats to the current output stream, usually the display.

**Example**

```
var v;
v := 05678;
putnum(HEX, v);     // print the number as hex 4 digits
putnum(BIN, v);     // print the number as binary 16 digits
```

## 2.5.5 print(...)

| Syntax | print(...); |
|---|---|

**4DGL** has a versatile **print(...)** statement for formatting numbers and strings. In it's simplest form, print will simply print a number as can be seen below:

```
myvar := 100;
print(myvar);
```

This will print **100** to the current output device (usually the display in TEXT mode). Note that if you wish to add a string anywhere within a print(...) statement, just place a quoted string expression and you will be able to mix strings and numbers in a variety of formats. See the following example.

```
print("the value of myvar is :- ", myvar, "and its 8bit binary representation is:-", [BIN8]myvar);
```

*** Refer the the table in putnum(..) for all the numeric representations available.**

The print(...) statement will accept directives passed in square brackets to make it print in various ways, for instance, if you wish to print a number in 4 digit hex, use the **[HEX4]** directive placed in front of the variable to be displayed within the print statement. See the following example.

```
print("myvar as a 4 digit HEX number is :- ", [HEX4]myvar);
```

Note that there are 2 print directives that are not part of the numeric set and will be explained separately. these are the **[STR]** and **[CHR]** directives.

The **[STR]** directive expects a string pointer to follow:

```
s := "Hello World"; // assign a string constant to s
print("Var 's' points to a string constant at address", s ," which is", [STR] s);
```

The **[CHR]** directive prints the character value of a variable.

```
print("The third character of the string is '", [CHR] *(s+2));
```

also

```
print("The value of 'myvar' as an ASCII charater is '", [CHR] myvar);
```

Note that you can freely mix string pointers, strings, variables and expressions within a print statement. print(...) can also use the to(...) function to redirect it's output to a different output device other than the screen using the function (refer to the **to(...)** statement for further examples).

## 2.5.6 to(outstream)

**Syntax** **to(outstream);**

**Arguments** **outstream**

**outstream** A variable or constant specifying the destination for the **putch**, **putstr, putnum** and **print** functions.

| Predefined Name | Constant | putch(), putstr(), putnum(), print() redirection |
|---|---|---|
| **APPEND** | 0x0000 | Output is directed to the same stream that was previously assigned. Output is appended to user array if previous redirection was to an array. |
| **COM0** | 0xFF04 | Output is redirected to the **COM** (serial) port. |
| **TEXT** | 0xFF08 | Output is directed to the **screen** (default). |
| **MDA** | 0xFF10 | Output is directed to the **SD/SDHC** or **FLASH** media. |
| **(memory pointer)** | 0x102 < 0x3FF | Output is redirect to the **memory** pointer argument. |

**Returns** **nothing**

**Description** **to()** sends the printed output to destinations other than the screen. Normally, print just sends its output to the display in **TEXT** mode which is the default, however, the output from print can be sent to **COM0**, and **MDA** (media) 'streams'. The **to(...)** function can also stream to a memory array . Note that once the **to(...)** function has taken effect, the stream reverts back to the default stream which is **TEXT** as soon as **putch**, **putstr**, **putnum** or **print** has completed its action. The **APPEND** argument is used to send the printed output to the same place as the previous redirection. This is most useful for building string arrays, or adding sequential data to a media stream.

**Example**

```
//====================================================
// Example #1 - putstr redirection
//====================================================
var buf[10];                  // a buffer that will hold up to 20
bytes/chars
var s;                        // a var for use as a pointer
to(buf); putstr("ONE ");      // redirect putstr to the buffer
to(APPEND); putstr("TWO ");   // and add a couple more items
to(APPEND); putstr("THREE\n");
putstr(buf);                  // print the result

while (media_Init()==0);      // wait if no SD/SDHC card detected
media_SetSector(0, 2);        // at sector 2
//media_SetAdd(0, 1024);      // (alternatively, use media_SetAdd(),
                              // lower 9 bits ignored).
to(MDA); putstr("Hello World");   // now write a ascii test string
media_WriteByte('A');             // write a further 3 bytes
media_WriteByte('B');
media_WriteByte('C');
to(MDA); putstr(buf);         // write the buffer we prepared earlier
```

```
media_WriteByte(0);             // terminate with ASCII zero
media_Flush();
media_SetAdd(0, 1024);          // reset the media address
while(char:=media_ReadByte())
    to(COM0); putch(char); // print the stored string to the COM port
wend
repeat forever
```

### 2.5.7 charwidth('char')

| **Syntax** | **charwidth('char');** | |
|---|---|---|
| | | |
| **Arguments** | **'char'** | |
| | **'char'** | The ascii character for the width calculation. |
| | | |
| **Returns** | **width** | |
| | **width** | Returns the width of a single character in pixel units. |
| | | |
| **Description** | **charwidth** is used to calculate the width in pixel units for a string, based on the currently selected font. The font can be proportional or mono-spaced. If the total width of the string exceeds 255 pixel units, the function will return the 'wrapped' (modulo 8) value. | |
| | | |
| **Example** | | |

```
//==================================================
// Example
//==================================================
str := "HELLO\nTHERE";   // note that this string spans 2 lines due
                         // to the \n.
width := strwidth(str);  // get the width of the string, this will
                         // also capture the height.
height := strheight();   // note, invoking strwidth also calcs height
                         // which we can now read.
// The string above spans 2 lines, strheight() will calculate height
// correctly for multiple lines.
len := strlen(str);      // the strlen() function returns the number
                         // of characters in a string.
print("\nLength=",len);  // NB:- the \n in "HELLO\nTHERE" is counted
                         // as a character.
txt_FontID(MS_SanSerif8x12); // select this font
w := charwidth('W');         // get a characters width
h := charheight('W');        // and height
txt_FontID(0);               // back to default font
print ("\n'W' is " ,w, " pixels wide"); // show width of a character
                                        // 'W' in pixel units.
print ("\n'W' is " ,h, " pixels high"); // show height of a character
                                        // 'W' in pixel units.
```

### 2.5.8 charheight('char')

| | | |
|---|---|---|
| **Syntax** | **charheight('char');** | |
| | | |
| **Arguments** | **'char'** | |
| | **'char'** | The ascii character for the height calculation. |
| | | |
| **Returns** | **width** | |
| | **width** | Returns the height of a single character in pixel units. |
| | | |
| **Description** | **charheight** is used to calculate the height in pixel units for a string, based on the currently selected font. The font can be proportional or mono-spaced. | |
| | | |
| **Example** | `See example in charwidth()` | |

### 2.5.9 strwidth(pointer)

| | | |
|---|---|---|
| **Syntax** | **strlen(pointer);** | |
| | | |
| **Arguments** | **pointer** | |
| | **pointer** | The pointer to a zero (0x00) terminated string. |
| | | |
| **Returns** | **width** | |
| | **width** | Returns the width of a string in pixel units. |
| | | |
| **Description** | **strwidth** returns the width of a zero terminated string in pixel units. Note that any string constants declared in your program are automatically terminated with a zero as an end marker by the compiler. Any string that you create in the DATA section or MEM section must have a zero added as a terminator for this function to work correctly. | |
| | | |
| **Example** | `See example in charwidth()` | |

### 2.5.10 strheight()

| | | |
|---|---|---|
| **Syntax** | **strlen(pointer);** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **height** | |
| | **height** | Returns the height of a string in pixel units. |
| | | |
| **Description** | **strheight** returns the height of a zero terminated string in pixel units. The strwidth function must be called first which makes available width and height. Note that any string constants declared in your program are automatically terminated with a zero as an end marker by the compiler. Any string that you create in the DATA section or MEM section must have a zero added as a terminator for this function to work correctly. | |
| | | |
| **Example** | `See example in charwidth()` | |

### 2.5.11 strlen(pointer)

| Syntax | strlen(pointer); | |
|---|---|---|
| | | |
| Arguments | pointer | |
| | pointer | The pointer to a zero (0x00) terminated string. |
| | | |
| Returns | length | |
| | length | Returns the length of a string in character units. |
| | | |
| Description | **strlen** returns the length of a zero terminated string in character units. Note that any string constants declared in your program are automatically terminated with a zero as an end marker by the compiler. Any string that you create in the DATA section or MEM section must have a zero added as a terminator for this function to work correctly. | |
| | | |
| Example | `See example in charwidth()` | |

## 2.5.12 txt_Set(function, value)

| Syntax | txt_Set(function, value); | |
|---|---|---|
| | | |
| **Arguments** | **function, value** | |
| | **function** | The function number determines the required action for various text control functions. Usually a constant, but can be a variable, array element, or expression. There are pre-defined constants for each of the functions. |
| | **value** | A variable, array element, expression or constant holding a value for the selected function. |
| | | |
| **Returns** | **value** | |
| | **value** | Returns Previous value before change is made |
| | | |
| **Description** | Given a function number and a value, set the required text control parameter, such as size, colour, and other formatting controls. This function is extremely useful in a loop to select multiple parameters from a data statement or a control array. Note also that each function available for txt_Set has a single parameter 'shortcut' function that has the same effect.<br>(see the **Single parameter short-cuts for the txt_Set functions** next page) | |

| **function** | | | **value** |
|---|---|---|---|
| **#** | **Predefined Name** | **Description** | |
| 0 | TEXT_COLOUR | Set the text foreground colour | Colour 0-65535 |
| 1 | TEXT_HIGHLIGHT | Set the text background colour | Colour 0-65535 |
| 2 | FONT_ID | Set the required font. FONT1 or SYSTEM is default fonts.<br>**Note:** The value could be the name of a custom font included in a users program in a data statement. See examples in the 4DGL Workshop3 IDE. | FONT1 or SYSTEM |
| 3 | TEXT_WIDTH | Set the text width multiplier | 1 to 16 (Default =1) |
| 4 | TEXT_HEIGHT | Set the text height multiplier. | 1 to 16 (Default =1) |
| 5 | TEXT_XGAP | Set the pixel gap (in pixel units) between characters. | 0 to 32 (Default = 0) |
| 6 | TEXT_YGAP | Set the pixel gap (in pixel units) between lines. | 0 to 32 (Default = 0) |
| 7 | TEXT_PRINTDELAY | Set the delay between character printing | (Default 0msec) |
| 8 | TEXT_OPACITY | Selects whether or not the 'background' pixels are drawn (default mode is OPAQUE) | 0 or TRANSPARENT<br>1 or OPAQUE |
| 9 | TEXT_BOLD | Embolden text | 0 or 1 (ON or OFF) |
| 10 | TEXT_ITALIC | Italicise text | 0 or 1 (ON or OFF) |
| 11 | TEXT_INVERSE | Inverted text | 0 or 1 (ON or OFF) |
| 12 | TEXT_UNDERLINED | Underlined text | 0 or 1 (ON or OFF) |
| 13 | TEXT_ATTRIBUTES | Control of functions 9,10,11,12 grouped<br>(bits can be combined by using logical 'or' of bits)<br>**Note:** bits 0-3 and 8-15 are reserved | 16 or BOLD<br>32 or ITALIC<br>64 or INVERSE<br>128 or UNDERLINED |

Single parameter short-cuts for the txt_Set(..) functions

| **Function Syntax** | **Function Action** | **value** |
|---|---|---|
| **txt_FGcolour()** | Set the text foreground colour | Colour 0-65535 |
| **txt_BGcolour()** | Set the text background colour | Colour 0-65535 |
| **txt_FontID(id)** | Set the required font.<br>FONT1 or SYSTEM is default fonts.<br>**Note:** The value could be the name of a custom font included in a users program in a data statement. See examples in the 4DGL Workshop3 IDE. | FONT1 or SYSTEM |
| **txt_Width(multiplier)** | Set the text width multiplier. | 1 to 16 (Default =1) |
| **txt_Height(multiplier)** | Set the text height multiplier. | 1 to 16 (Default =1) |
| **txt_Xgap(pixelcount)** | Set the pixel gap (in pixel units) between characters. | 0 to 32 (Default = 0) |
| **txt_Ygap(pixelcount)** | Set the pixel gap (in pixel units) between lines. | 0 to 32 (Default = 0) |
| **txt_Delay(millisecs)** | Set the delay between character printing | (Default 0msec) |
| **txt_Opacity(mode)** | Selects whether or not the 'background' pixels are drawn (default mode is OPAQUE). | 0 or TRANSPARENT<br>1 or OPAQUE |
| **txt_Bold(mode)** | Embolden text. | 0 or 1 (ON or OFF) |
| **txt_Italic(mode)** | Italic text. | 0 or 1 (ON or OFF) |
| **txt_Inverse(mode)** | Inverted text. | 0 or 1 (ON or OFF) |
| **txt_Underlined(mode)** | Underlined text. | 0 or 1 (ON or OFF) |
| **txt_Attributes(value)** | Control of functions 9, 10, 11, 12 grouped<br>(bits can be combined by using logical 'OR' of bits)<br>**Note:** bits 0-3 and 8-15 are reserved. | 16 or BOLD<br>32 or ITALIC<br>64 or INVERSE<br>128 or UNDERLINED |
| **Note:** All shortcut commands return Previous value before change is made. | | |

## 2.6 Graphics Functions

**Summary of Functions in this section:**

- gfx_Cls()
- gfx_ChangeColour(oldColour, newColour)
- gfx_Circle(x, y, radius, colour)
- gfx_CircleFilled(x, y, radius, colour)
- gfx_Line(x1, y1, x2, y2, colour)
- gfx_Hline(y, x1, x2, colour)
- gfx_Vline(x, y1, y2, colour)
- gfx_Rectangle(x1, y1, x2, y2, colour)
- gfx_RectangleFilled(x1, y1, x2, y2, colour)
- gfx_Polyline(n, vx, vy, colour)
- gfx_Polygon(n, vx, vy, colour)
- gfx_Triangle(x1, y1, x2, y2, x3, y3, colour)
- gfx_Dot()
- gfx_Bullet(radius)
- gfx_OrbitInit(&x_dest, &y_dest)
- gfx_Orbit(angle, distance)
- gfx_PutPixel(x, y, colour)
- gfx_GetPixel(x, y)
- gfx_MoveTo(xpos, ypos)
- gfx_MoveRel(xoffset, yoffset)
- gfx_LineTo(xpos, ypos)
- gfx_LineRel(xpos, ypos)
- gfx_BoxTo(x2, y2)
- gfx_SetClipRegion()
- gfx_ClipWindow(x1, y1, x2, y2)
- gfx_FocusWindow()
- rect_Intersect(&rect1, &rect2)
- rect_Within(&rect1, &rect2)
- gfx_Set(function, value)

  **gfx_Set shortcuts:**
  - gfx_PenSize(mode)
  - gfx_BGcolour(colour)
  - gfx_ObjectColour(colour)
  - gfx_Clipping(mode)
  - gfx_TransparentColour(colour)
  - gfx_Transparency(mode)
  - gfx_FrameDelay(delay)
  - gfx_ScreenMode(mode)
  - gfx_OutlineColour(colour)
  - gfx_Contrast(value)
  - gfx_LinePattern(pattern)
  - gfx_ColourMode(mode)

### 2.6.1 gfx_Cls()

| | |
|---|---|
| **Syntax** | **gfx_Cls();** |
| | |
| **Arguments** | **none** |
| | |
| **Returns** | **nothing** |
| | |
| **Description** | Clear the screen using the current background colour |
| | |
| **Example** | `gfx_BGcolour(DARKGRAY);`<br>`gfx_Cls();`<br><br>This example clears the entire display using colour DARKGRAY |

## 2.6.2 gfx_ChangeColour(oldColour, newColour)

| | | |
|---|---|---|
| **Syntax** | **gfx_ChangeColour(oldColour, newColour);** | |
| **Arguments** | **oldColour, newColour** | |
| | **oldColour** | specifies the sample colour to be changed within the clipping window. |
| | **newColour** | specifies the new colour to change all occurrences of old colour within the clipping window. |
| | The arguments can be a variable, array element, expression or constant | |
| **Returns** | **nothing** | |
| **Description** | Changes all **oldColour pixels** to **newColour** within the clipping area. | |

**Example**

```
func main()
   txt_Width(3);
   txt_Height(5);
   gfx_MoveTo(8,20);
   print("TEST");            // print the string
   gfx_SetClipRegion();      // force clipping area to extents of text
                             // just printed.
   gfx_ChangeColour(BLACK, RED); // test change of background colour

   repeat forever
endfunc
```

This example prints a test string, forces the clipping area to the extent of the text that was printed, then changes the background colour.

### 2.6.3 gfx_Circle(x, y, radius, colour)

| | | |
|---|---|---|
| **Syntax** | **gfx_Circle(x, y, rad, colour);** | |
| | | |
| **Arguments** | **x, y, rad, colour** | |
| | **x, y** | specifies the center of the circle. |
| | **rad** | specifies the radius of the circle. |
| | **colour** | specifies the colour of the circle. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Draws a circle with centre point x1, y1 with radius r using the specified colour.<br>NB: The default **PEN_SIZE** is set to **OUTLINE**, however, if **PEN_SIZE** is set to **SOLID**, the circle will be drawn filled, if **PEN_SIZE** is set to **OUTLINE**, the circle will be drawn as an outline. If the circle is drawn as **SOLID**, the outline colour can be specified with **gfx_OutlineColour(...).** If **OUTLINE_COLOUR** is set to 0, no outline is drawn. | |
| | | |
| **Example** | `// assuming PEN_SIZE is OUTLINE`<br>`gfx_Circle(50,50,30, 0x001F);`<br>This example draws a BLUE circle outline centred at x=50, y=50 with a radius of 30 pixel units. | |

### 2.6.4 gfx_CircleFilled(x, y, radius, colour)

| | | |
|---|---|---|
| **Syntax** | **gfx_CircleFilled(x, y, rad, colour);** | |
| | | |
| **Arguments** | **x, y, rad, colour** | |
| | **x, y** | specifies the center of the circle. |
| | **rad** | specifies the radius of the circle. |
| | **colour** | specifies the fill colour of the circle. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Draws a **SOLID** circle with centre point x1, y1 with radius using the specified colour.<br>The outline colour can be specified with **gfx_OutlineColour(...).** If **OUTLINE_COLOUR** is set to 0, no outline is drawn.<br>NB:- The **PEN_SIZE** is ignored, the circle is always drawn **SOLID.** | |
| | | |
| **Example** | `gfx_OutlineColour(0xFFE0);`<br>`gfx_CircleFilled(25,25,10, 0xF800);`<br>This example draws a filled RED circle with a YELLOW outline at x=25, y=25 with a radius of 10 pixel units. | |

### 2.6.5 gfx_Line(x1, y1, x2, y2, colour)

| | | |
|---|---|---|
| **Syntax** | **gfx_Line(x1, y1, x2, y2, colour);** | |
| | | |
| **Arguments** | **x1, y1, x2, y2, colour** | |
| | **x1, y1** | specifies the starting coordinates of the line. |
| | **x2, y2** | specifies the ending coordinates of the line. |
| | **colour** | specifies the colour of the line. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Draws a line from x1,y1 to x2,y2 using the specified colour. The line is drawn using the current object colour. The current origin is not altered. The line may be tessellated with the **gfx_LinePattern(...)** function. | |
| | | |
| **Example** | `gfx_Line(100, 100, 10, 10, 0xF800);`<br>This example draws a RED line from x1=10, y1=10 to x2=100, y2=100 | |

### 2.6.6 gfx_Hline(y, x1, x2, colour)

| | | |
|---|---|---|
| **Syntax** | **gfx_Hline(y, x1, x2, colour);** | |
| | | |
| **Arguments** | **y, x1, x2, colour** | |
| | **y** | specifies the vertical position of the horizontal line. |
| | **x1, x2** | specifies the horizontal end points of the line. |
| | **colour** | specifies the colour of the horizontal line. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Draws a fast horizontal line from x1 to x2 at vertical co-ordinate y using colour. | |
| | | |
| **Example** | `gfx_Hline(50, 10, 80, 0xF800);`<br>This example draws a fast RED horizontal line at y=50, from x1=10 to x2=80 | |

### 2.6.7 gfx_Vline(x, y1, y2, colour)

| | | |
|---|---|---|
| **Syntax** | **gfx_Vline(x, y1, y2, colour);** | |
| | | |
| **Arguments** | **x, y1, y2, colour** | |
| | **x** | specifies the horizontal position of the vertical line. |
| | **y1, y2** | specifies the vertical end points of the line. |
| | **colour** | specifies the colour of the vertical line. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Draws a fast vertical line from y1 to y2 at horizontal co-ordinate x using colour. | |
| | | |
| **Example** | `gfx_Vline(20, 30, 70, 0xF800);`<br>This example draws a fast RED vertical line at x=20, from y1=30 to y2=70 | |

### 2.6.8 gfx_Rectangle(x1, y1, x2, y2, colour)

| | | |
|---|---|---|
| **Syntax** | **gfx_Rectangle(x1, y1, x2, y2, colour);** | |
| | | |
| **Arguments** | **x1, y1, x2, y2, colour** | |
| | **x1, y1** | specifies the top left corner of the rectangle. |
| | **x2, y2** | specifies the bottom right corner of the rectangle. |
| | **colour** | specifies the colour of the rectangle. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Draws a rectangle from x1, y1 to x2, y2 using the specified colour. The line may be tessellated with the **gfx_LinePattern(...)** function.<br>NB: The default **PEN_SIZE** is set to **OUTLINE**, however, if **PEN_SIZE** is set to **SOLID**, the rectangle will be drawn filled, if **PEN_SIZE** is set to **OUTLINE**, the rectangle will be drawn as an outline. If the rectangle is drawn as **SOLID**, the outline colour can be specified with **gfx_OutlineColour(...). If OUTLINE_COLOUR** is set to 0, no outline is drawn. The outline may be tessellated with the **gfx_LinePattern(...)** function. | |
| | | |
| **Example** | `// assuming PEN_SIZE is OUTLINE`<br>`gfx_Rectangle(10, 10, 30, 30, 0x07E0);`<br>This example draws a GREEN rectangle from x1=10, y1=10 to x2=30, y2=30 | |

### 2.6.9 gfx_RectangleFilled(x1, y1, x2, y2, colour)

| | | |
|---|---|---|
| **Syntax** | **gfx_RectangleFilled(x1, y1, x2, y2, colour);** | |
| | | |
| **Arguments** | **x1, y1, x2, y2, colour** | |
| | **x1, y1** | specifies the top left corner of the rectangle. |
| | **x2, y2** | specifies the bottom right corner of the rectangle. |
| | **colour** | specifies the colour of the rectangle. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Draws a **SOLID** rectangle from x1, y1 to x2, y2 using the specified colour. The line may be tessellated with the **gfx_LinePattern(...)** function.<br>The outline colour can be specified with **gfx_OutlineColour(...). If OUTLINE_COLOUR** is set to 0, no outline is drawn. The outline may be tessellated with the **gfx_LinePattern(...)** function.<br>NB:- The **PEN_SIZE** is ignored, the rectangle is always drawn **SOLID.** | |
| | | |
| **Example** | `gfx_OutlineColour(0xFFE0);`<br>`gfx_RectangleFilled(30,30,80,80, 0xF800);`<br>This example draws a filled RED rectangle with a YELLOW outline from x1=30,y1=30 to x2=80,y2=80 | |

### 2.6.10 gfx_Polyline(n, vx, vy, colour)

| | | |
|---|---|---|
| **Syntax** | **gfx_Polyline(n, vx, vy, colour);** | |
| | | |
| **Arguments** | **n, vx, vy, colour** | |
| | **n** | specifies the number of elements in the x and y arrays specifying the vertices for the polyline. |
| | **vx** | specifies the addresses of the storage of the array of elements for the x coordinates of the vertices. |
| | **vy** | specifies the addresses of the storage of the array of elements for the y coordinates of the vertices. |
| | **colour** | Specifies the colour for the lines |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Plots lines between points specified by a pair of arrays using the specified colour. The lines may be tessellated with the **gfx_LinePattern(...)** function. gfx_Polyline can be used to create complex raster graphics by loading the arrays from serial input or from MEDIA with very little code requirement. | |
| | | |
| **Example** | | |

```
#inherit "4DGL_16bitColours.fnc"

var vx[20], vy[20];

func main()
    vx[0] := 36; vy[0] := 110;
    vx[1] := 36; vy[1] := 80;
    vx[2] := 50; vy[2] := 80;
    vx[3] := 50; vy[3] := 110;

    vx[4] := 76; vy[4] := 104;
    vx[5] := 85; vy[5] := 80;
    vx[6] := 94; vy[6] := 104;

    vx[7] := 76; vy[7] := 70;
    vx[8] := 85; vy[8] := 76;
    vx[9] := 94; vy[9] := 70;

    vx[10] := 110; vy[10] := 66;
    vx[11] := 110; vy[11] := 80;
    vx[12] := 100; vy[12] := 90;
    vx[13] := 120; vy[13] := 90;
    vx[14] := 110; vy[14] := 80;

    vx[15] := 101; vy[15] := 70;
    vx[16] := 110; vy[16] := 76;
    vx[17] := 119; vy[17] := 70;
    // house
    gfx_Rectangle(6,50,66,110,RED);         // frame
    gfx_Triangle(6,50,36,9,66,50,YELLOW);   // roof
    gfx_Polyline(4, vx, vy, CYAN);          // door

    // man
    gfx_Circle(85, 56, 10, BLUE);           // head
    gfx_Line(85, 66, 85, 80, BLUE);         // body
    gfx_Polyline(3, vx+4, vy+4, CYAN);      // legs
    gfx_Polyline(3, vx+7, vy+7, BLUE);      // arms

    // woman
    gfx_Circle(110, 56, 10, PINK);          // head
    gfx_Polyline(5, vx+10, vy+10, BROWN);   // dress
    gfx_Line(104, 104, 106, 90, PINK);      // left arm
    gfx_Line(112, 90, 116, 104, PINK);      // right arm
    gfx_Polyline(3, vx+15, vy+15, SALMON);  // dress

    repeat forever

endfunc
```

This example draws a simple scene

### 2.6.11 gfx_Polygon(n, vx, vy, colour)

| **Syntax** | **gfx_Polygon(n, vx, vy, colour);** | |
|---|---|---|
| | | |
| **Arguments** | **n, vx, vy, colour** | |
| | **n** | specifies the number of elements in the x and y arrays specifying the vertices for the polygon. |
| | **vx** | specifies the addresses of the storage of the array of elements for the x coordinates of the vertices. |
| | **vy** | specifies the addresses of the storage of the array of elements for the y coordinates of the vertices. |
| | **colour** | Specifies the colour for the polygon |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Plots lines between points specified by a pair of arrays using the specified colour. The last point is drawn back to the first point, completing the polygon. The lines may be tessellated with the **gfx_LinePattern(...)** function. gfx_Polygon can be used to create complex raster graphics by loading the arrays from serial input or from MEDIA with very little code requirement. | |
| | | |

**Example**

```
var vx[7], vy[7];

func main()
    vx[0] := 10; vy[0] := 10;
    vx[1] := 35; vy[1] := 5;
    vx[2] := 80; vy[2] := 10;
    vx[3] := 60; vy[3] := 25;
    vx[4] := 80; vy[4] := 40;
    vx[5] := 35; vy[5] := 50;
    vx[6] := 10; vy[6] := 40;
    gfx_Polygon(7, vx, vy, RED);

    repeat forever
endfunc
```

This example draws a simple polygon

### 2.6.12 gfx_Triangle(x1, y1, x2, y2, x3, y3, colour)

| | | |
|---|---|---|
| **Syntax** | **gfx_Triangle(x1, y1, x2, y2, x3, y3, colour);** | |
| | | |
| **Arguments** | **x1, y1, x2, y2, x3, y3, colour** | |
| | **x1, y1** | specifies the first vertices of the triangle. |
| | **x2, y2** | specifies the second vertices of the triangle. |
| | **x3, y3** | specifies the third vertices of the triangle. |
| | **colour** | Specifies the colour for the triangle. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Draws a triangle outline between vertices x1,y1 , x2,y2 and x3,y3 using the specified colour. The line may be tessellated with the **gfx_LinePattern(...)** function. | |
| | | |
| **Example** | `gfx_Triangle(10,10,30,10,20,30,0xFFE0);`<br>This example draws a CYAN triangular outline with vertices at 10,10 30,10 20,30 | |

### 2.6.13 gfx_Dot()

| | |
|---|---|
| **Syntax** | **gfx_Dot();** |
| | |
| **Arguments** | **none** |
| | |
| **Returns** | **nothing** |
| | |
| **Description** | Draws a **pixel** at at the current origin using the current object colour. |
| | |
| **Example** | gfx_MoveTo(40,50);<br>gfx_ObjectColour(0xF800);<br>gfx_Dot();<br><br>This example draws a RED pixel at 40,50 |

## 2.6.14 gfx_Bullet(radius)

| | | |
|---|---|---|
| **Syntax** | **gfx_Bullet(radius);** | |
| | | |
| **Arguments** | **radius** | |
| | **rad** | specifies the radius of the bullet. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Draws a **circle** or 'bullet point' with radius ***r*** at at the current origin using the current object colour.<br>**Note**: The default **PEN_SIZE** is set to **OUTLINE**, however, if **PEN_SIZE** is set to **SOLID**, the circle will be drawn filled, if **PEN_SIZE** is set to **OUTLINE**, the circle will be drawn as an outline. If the circle is drawn as **SOLID**, the outline colour can be specified with **gfx_OutlineColour(...).** | |
| | | |
| **Example** | `// assuming PEN_SIZE is TRANSPARENT`<br>`// and OBJECT_COLOUR is WHITE`<br><br>`gfx_MoveTo(50,50);`<br>`gfx_Bullet(5);`<br><br>This example draws a WHITE circle outline at the current origin with a radius of 5 pixel units. | |

## 2.6.15 gfx_OrbitInit(&x_dest, &y_dest)

| | | |
|---|---|---|
| **Syntax** | **gfx_OrbitInit(&x_dest, &y_dest);** | |
| | | |
| **Arguments** | **&x_dest, &y_dest** | |
| | **&x_dest,**<br>**&y_dest** | specifies the addresses of the storage locations for the orbit calculation. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Sets up the internal pointers for the **gfx_Orbit(..)** result variables. The &***x_orb*** and &***y_orb*** parameters are the addresses of the variables or array elements that are used to store the result from the **gfx_Orbit(..)** function. | |
| | | |
| **Example** | `var targetX, targetY;`<br>`gfx_OrbitInit(&targetX, &targetY);`<br><br>This example sets the variables that will receive the result from a **gfx_Orbit(..)** function call | |

### 2.6.16 gfx_Orbit(angle, distance)

**Syntax** **gfx_Orbit(angle, distance);**

**Arguments** **angle, distance**

| | |
|---|---|
| **angle** | specifies the angle from the origin to the remote point. The angle is specified in degrees. |
| **distance** | specifies the distance from the origin to the remote point in pixel units. |

The arguments can be a variable, array element, expression or constant

**Returns** **nothing**

**Note**: result is stored in the variables that were specified with the **gfx_OrbitInit(..)** function.

**Description** Sets Prior to using this function, the destination address of variables for the calculated coordinates must be set using the **gfx_OrbitInit(..)** function. The **gfx_Orbit(..)** function calculates the x, y coordinates of a distant point relative to the current origin, where the only known parameters are the ***angle*** and the ***distance*** from the current origin. The new coordinates are calculated and then placed in the destination variables that have been previously set with the **gfx_OrbitInit(..)** function.

**Example**

```
var targetX, targetY;
gfx_OrbitInit(&targetX, &targetY);
gfx_MoveTo(30, 30);
gfx_Bullet(5)      // mark the start point with a small WHITE circle
gfx_Orbit(30, 50); // calculate a point 50 pixels away from origin at
                   // 30 degrees
gfx_CircleFilled(targetX,targetY,3,0xF800); // mark the target point
                                            // with a RED circle
```

See example comments for explanation.

## 2.6.17 gfx_PutPixel(x, y, colour)

| | | |
|---|---|---|
| **Syntax** | **gfx_PutPixel(x, y, colour);** | |
| | | |
| **Arguments** | **x, y, colour** | |
| | **x, y** | specifies the screen coordinates of the pixel. |
| | **colour** | Specifies the colour of the pixel. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Draws a pixel at position x,y using the specified colour. | |
| | | |
| **Example** | `gfx_PutPixel(32, 32, 0xFFFF);`<br>This example draws a WHITE pixel  at x=32, y=32 | |

### 2.6.18 gfx_GetPixel(x, y)

| | | |
|---|---|---|
| **Syntax** | **gfx_GetPixel(x, y);** | |
| | | |
| **Arguments** | **x, y** | |
| | **x, y** | specifies the screen coordinates of the pixel colour to be returned. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **colour** | |
| | colour | The 8 or 16bit colour of the pixel (default 16bit). |
| | | |
| **Description** | Reads the colour value of the pixel at position x,y. | |
| | | |
| **Example** | `gfx_PutPixel(20, 20, 1234);`<br>`r := gfx_GetPixel(20, 20);`<br>`print(r);`<br>This example prints 1234, the colour of the pixel that was previously placed. | |

### 2.6.19 gfx_MoveTo(xpos, ypos)

| | | |
|---|---|---|
| **Syntax** | **gfx_MoveTo(xpos, ypos);** | |
| | | |
| **Arguments** | **xpos, ypos** | |
| | **xpos** | specifies the horizontal position of the new origin. |
| | **ypos** | specifies the vertical position of the new origin. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Moves the origin to a new position. | |
| | | |
| **Example** | `gfx_MoveTo(10, 20);`<br>`gfx_Dot();`<br>This example moves the origin to x=10, y=20 and draws a pixel. | |

## 2.6.20 gfx_MoveRel(xoffset, yoffset)

| | | |
|---|---|---|
| **Syntax** | **gfx_MoveRel(xoffset, yoffset);** | |
| | | |
| **Arguments** | **xoffset, yoffset** | |
| | **xoffset** | specifies the horizontal offset of the new origin. |
| | **yoffset** | specifies the vertical offset of the new origin. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Moves the origin to a new position relative to the old position. | |
| | | |
| **Example** | `gfx_MoveTo(10, 20);`<br>`gfx_MoveRel(-5, -3);`<br>`gfx_Dot();`<br>This example draws a pixel using the current object colour at x=5, y=17 | |

### 2.6.21 gfx_LineTo(xpos, ypos)

| | | |
|---|---|---|
| **Syntax** | **gfx_LineTo(xpos, ypos);** | |
| | | |
| **Arguments** | **xpos, ypos** | |
| | **xpos** | specifies the horizontal position of the line end as well as the new origin. |
| | **ypos** | specifies the vertical position of the line end as well as the new origin. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Draws a line from the current origin to a new position. The Origin is then set to the new position. The line is drawn using the current object colour. The line may be tessellated with the **gfx_LinePattern(...)** function. | |
| | | |
| **Example** | `gfx_MoveTo(10, 20);`<br>`gfx_LineTo(60, 70);`<br><br>This example draws a line using the current object colour between x1=10,y1=20 and x2=60,y2=70. The new origin is now set at x=60,y=70. | |

## 2.6.22 gfx_LineRel(xpos, ypos)

| | | |
|---|---|---|
| **Syntax** | **gfx_LineRel(xpos, ypos);** | |
| | | |
| **Arguments** | **xpos, ypos** | |
| | **xpos** | specifies the horizontal end point of the line. |
| | **ypos** | specifies the vertical end point of the line. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Draws a line from the current origin to a new position. The line is drawn using the current object colour. The current origin is not altered. The line may be tessellated with the **gfx_LinePattern(...)** function. | |
| | | |
| **Example** | `gfx_LinePattern(0b1100110011001100);`<br>`gfx_MoveTo(10, 20);`<br>`gfx_LineRel(50, 50);`<br><br>This example draws a tessellated line using the current object colour between 10,20 and 50,50.<br>**Note:** that **gfx_LinePattern(0);** must be used after this to return line drawing to normal solid lines. | |

## 2.6.23 gfx_BoxTo(x2, y2)

| | | |
|---|---|---|
| **Syntax** | **gfx_BoxTo(x2, y2);** | |
| | | |
| **Arguments** | **x2, y2** | |
| | **x2,y2** | specifies the diagonally opposed corner of the rectangle to be drawn, the top left corner (assumed to be x1, y1) is anchored by the current origin. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Draws a rectangle from the current origin to the new point using the current object colour. The top left corner is anchored by the current origin (x1, y1), the bottom right corner is specified by x2, y2.<br>**Note**: The default **PEN_SIZE** is set to **OUTLINE**, however, if **PEN_SIZE** is set to **SOLID**, the rectangle will be drawn filled, if **PEN_SIZE** is set to **OUTLINE**, the rectangle will be drawn as an outline. If the circle is drawn as **SOLID**, the outline colour can be specified with **gfx_OutlineColour(...). If OUTLINE_COLOUR** is set to 0, no outline is drawn. | |
| | | |

**Example**

```
gfx_MoveTo(40,40);
n := 10;
while (n--)
   gfx_BoxTo(50,50);
   gfx_BoxTo(30,30);
wend
```

This example draws 2 boxes, anchored from the current origin.

## 2.6.24 gfx_SetClipRegion()

| | |
|---|---|
| **Syntax** | **gfx_SetClipRegion();** |
| | |
| **Arguments** | **none** |
| | |
| **Returns** | **nothing** |
| | |
| **Description** | Forces the clip region to the extent of the last text that was printed, or the last image that was shown. |
| | |
| **Example** | |

```
#constant NUMCOLOURS 6
var colour[NUMCOLOURS];
func main()
    var n,x,y,colr,x1,y1,x2,y2,w,h;
    colour[0]:=RED;     // the colour set for the random pixels
    colour[1]:=GREEN;
    colour[2]:=BLUE;
    colour[3]:=YELLOW;
    colour[4]:=CYAN;
    colour[5]:=MAGENTA;
    txt_Width(5); txt_Height(7);
    gfx_MoveTo(6,20);
    txt_Bold(ON);
    txt_FGcolour(1);           // start with a very dark blue
    print("TEST");             // print the string
    gfx_SetClipRegion();       // force clipping area to extents of
                               // text just printed
    x1:=peekB(CLIP_LEFT_POS);  // get the cliiping area to local vars
    y1:=peekB(CLIP_TOP_POS);
    x2:=peekB(CLIP_RIGHT_POS);
    y2:=peekB(CLIP_BOTTOM_POS);
    w:=x2-x1;                  // get the width and height
    h:=y2-y1;
    txt_MoveCursor(10,0);
    txt_FGcolour(SALMON);
    print("x1=",x1," y1=",y1,"\nx2=",x2," y2=",y2); //print the
                                                    //clipping region
    txt_FGcolour(GREEN);
    pause(1000);
    repeat
        if (!*TIMER0)                       // if timer has expired-
            *TIMER0 := 5000;                // reset the timer.
            colr := colour[n++%NUMCOLOURS]; // select new colour -
                                            // every 5 seconds.
            txt_MoveCursor(14,0);
            print([DEC5ZB] n);              // print n
       endif
       x:=ABS(RAND()%w) + x1;    // get random pixel position within
                                 // the clip region.
       y:=ABS(RAND()%h) + y1;
       if(gfx_GetPixel(x,y)) gfx_PutPixel(x,y, colr); // update any
                                                  // non black pixels
    forever
endfunc

```

This example prints a test string, forces the clipping area to the extent of the text that was printed, then changes the text  colour randomly, pixel by pixel.

### 2.6.25 gfx_ClipWindow(x1, y1, x2, y2)

| | | |
|---|---|---|
| **Syntax** | **gfx_ClipWindow(x1, y1, x2, y2);** | |
| | | |
| **Arguments** | **x1, y1, x2, y2** | |
| | **x1, y1** | specifies the horizontal and vertical position of the top left corner of the clipping window. |
| | **x2, y2** | specifies the horizontal and vertical position of the bottom right corner of the clipping window. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Specifies a clipping window region on the screen such that any objects and text placed onto the screen will be clipped and displayed only within that region. For the clipping window to take effect, "Clipping" setting must be enabled separately using **gfx_Set(CLIPPING, ON)** or the shortcut **gfx_Clipping(ON)**. | |
| | | |

**Example**

```
var n;
gfx_ClipWindow(10, 10, 50, 50 )
n := 5000;
while(n--)
    gfx_PutPixel(RAND()%100, RAND()%100, RAND());
wend
repeat forever
```

This example will draw 5000 random colour pixels, only the pixels within the clipping area will be visible

### 2.6.26 gfx_FocusWindow()

| | | |
|---|---|---|
| **Syntax** | **gfx_FocusWindow();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **pixel_count** | |
| | **pixel_count** | The pixel count of the selected area. |
| | | |
| **Description** | Sets the display hardware GRAM access registers to the clipping area ready for reading or writing. The function also returns the pixel count of the selected area. | |
| | | |

**Example**

```
// example #1
func main()
   var pixelcount;
   txt_Height(4);
   gfx_MoveTo(20,20);
   print("TEST");                  // print a string.
   gfx_SetClipRegion();            // force the clipping region to the
                                   // extent of the text.
   Pixelcount:= gfx_FocusWindow(); // get the count, focus on region.
   pause(1000);
   disp_BlitPixelFill(BLUE, pixelcount);  // fill the region.
   print(pixelcount, " pixels\n"); //show the pixel count of region.
   repeat forever
endfunc
```

The above example prints a test string, forces the clipping area to the extent of the text that was printed, then after a delay, fills the region with a colour. The count of pixels in the region is then shown.

## 2.6.27 gfx_SpriteSet(bitmaps, colours, palette)

| | | |
|---|---|---|
| **Syntax** | **gfx_SpriteSet(bitmaps, colours, palette);** | |
| | | |
| **Arguments** | **bitmaps, colours, palette** | |
| | **bitmaps** | See the description. |
| | **colours** | See the description. |
| | **palette** | See the description. |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | **3 sets of data are required by the sprite generator:-**<br>This function sets the internal pointers for the 3 parts.<br>**1]** The bitmaps for the sprites.<br>**2]** The colour lookup table (CLUT).<br>**3]** The 4 colour palettes.<br><br>**Sprite bitmap format:-**<br>Each sprite is 32 words long.<br>The first word (and subsequent even words) are sprite pixels 1-8 of the line.<br>The second word (and subsequent odd words) are sprite pixels 9-16 of the line etc.<br>The most significant pixel pair is the leftmost pixel.<br>The least significant pixel pair is the rightmost pixel.<br>Each pixel pair selects 1 of 4 colours from the selected pallette<br>Each palette has one of 4 colours that are 'wired' to the colour lookup table.<br>Each sprite can be displayed with a different palette, allowing colour cycling and other special effects. | |
| | | |

**Example**

```
Example sprite data, only 2 entry shown for clarity (128 max
available)

//===================================================================
// To make your code 'SpriteEditor friendly' yo must follow the
// following conventions...
//===================================================================

// The first comment line of each sprite bitmap is a description
// which is used by the sprite editor. Also, there is a naming
// convention used to identify the various data statement blocks to
// the SpriteEditor so it knows which data statements to use...

// Naming conventions:-
// 1] The sprite bitmap info must be stored as 'words' , the name
//    must end in _sprites, and each line must only contain 2 words
//    (1 line of pixels).
// 2] The colour lookup table must be stored as 'words' , each line
//    must have just one single word (colour), and its name must end
//    in _colors.

// 3] The palette must be stored as 'bytes', each line must have
//   just 4 bytes (1 palette entry)and its name must end in _palette.
//=====================================================================

// part 1] the bitmaps for the sprites

// #DATA
//      WORD mysprites_sprites
//  1) a box with a '+' in the middle
//      0xFFFF,0xFFFF,          // line 1       3333333333333333
//      0x0003,0xC000,          // line 2       3              3
//      0x0003,0xC000,          // line 3       3              3
//      0x0003,0xC000,          // line 4       3              3
//      0x4003,0xC001,          // line 5       3      11      3
//      0x4003,0xC001,          // line 6       3      11      3
//      0x4003,0xC001,          // line 7       3      11      3
//      0x5503,0xC055,          // line 8       3  1111111111  3
//      0x5503,0xC055,          // line 9       3  1111111111  3
//      0x4003,0xC001,          // line 10      3      11      3
//      0x4003,0xC001,          // line 11      3      11      3
//      0x4003,0xC001,          // line 12      3      11      3
//      0x0003,0xC000,          // line 13      3              3
//      0x0003,0xC000,          // line 14      3              3
//      0x0003,0xC000,          // line 15      3              3
//      0xFFFF,0xFFFF,          // line 16      3333333333333333

// 2) cherries
//      0x0000,0x0000,          // line 1
//      0x0000,0x0000,          // line 2
//      0x0000,0x0500,          // line 3                   11
//      0x0000,0x0550,          // line 4                 1111
//      0x0000,0x0045,          // line 5               11 1
//      0x4000,0x0040,          // line 6              1   1
//      0x1FC0,0x0010,          // line 7          3331   1
//      0xF7F0,0x0004,          // line 8         333133 1
//      0x3FF0,0x00F7,          // line 9         33333 3133
//      0xCFB0,0x03F7,          // line 10        3233 331333
//      0xCEF0,0x03FF,          // line 11        3323 333333
//      0xCFC0,0x03FE,          // line 12         333 323333
//      0xC000,0x03FB,          // line 13             332333
//      0x0000,0x00FF,          // line 14              3333
//      0x0000,0x0000,          // line 15
//      0x0000,0x0000           // line 16
//      ..........
// more bitmaps can follow.....
//       ..........
//  #END

// part 2] the colour lookup table (CLUT), with 13 example colour
//            entries (128 max available)

//#DATA
// word mycolours_colors
//      BLACK,                  // 0
//      RED,                    // 1
//      BROWN,                  // 2
//      PINK,                   // 3
//      CYAN,                   // 4
//      CYAN,                   // 5
//      BLUE,                   // 6
//      LIGHTSLATEGRAY,         // 7
//      ORANGE                  // 8
//      YELLOW,                 // 9
//      LIME,                   // 10
//      RED,                    // 11
//      WHITE                   // 12
//#END

// part 3] the palettes, each entry may have 4 colours.
//         The colours are selected from the CLUT

//#DATA
//  byte mypalette_palette
//      0,1,0,9, // black, red, black, yellow (box)
//      0,9,0,1, // black, yellow, black, red (box alternate colours)
//    0,10,12,1  // black, lime, white, red (for strawberry)
//#END
```

## 2.6.28 gfx_BlitSprite(spritenumber, palette, xpos, ypos, orientation)

| **Syntax** | **gfx_BlitSprite(spritenumber, palette, xpos, ypos, orientation);** | |
|---|---|---|
| | | |
| **Arguments** | **spritenumber, palette, xpos, ypos, orientation** | |
| | **spritenumber** | Select the required sprite name to be displayed |
| | **palette** | Select the required palette to use for the selected sprite |
| | **xpos** | specifies the horizontal and vertical position of the top left corner of the clipping window. |
| | **ypos** | specifies the horizontal and vertical position of the bottom right corner of the clipping window. |
| | **orientation** | NORTH<br>SOUTH<br>WEST<br>EAST<br>NORTH_MIRRORED<br>SOUTH_MIRRORED<br>WEST_MIRRORED<br>EAST_MIRRORED |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Places the required sprite bitmap at the origin xpos, ypos using the required 4 colour palette. orientation determines in which direction the sprite will be displayed. | |
| | | |
| **Example** | `gfx_BlitSprite(1,2,10,10,SOUTH);`<br>`// example show a cherry upside down - refer to demo programs`<br>`// for full explanation.` | |

## 2.6.29 rect_Intersect(&rect1, &rect2)

| | | |
|---|---|---|
| **Syntax** | **rect_Intersect(&rect1, &rect2);** | |
| | | |
| **Arguments** | **&rect1, &rect2** | |
| | **rect1** | Specifies the coordinates of rectangle 1. |
| | **rect2** | Specifies the coordinates of rectangle 2. |
| | The arguments should be an array of 4 words. | |
| | | |
| **Returns** | **Status** | |
| | Status | **1:** True<br>**0:** False. |
| | | |
| **Description** | Return true if any part of rect1 is within rect2. Each rectangle is an array of 4 words in the format.<br>element 0 = RECT_LEFT<br>element 1 = RECT_TOP<br>element 2 = RECT_RIGHT<br>element 3 = RECT_BOTTOM<br><br>This function is ideal for use as a collision detector | |
| | | |
| **Example** | `rect_Intersect(box1, box2);` | |

## 2.6.30 rect_Within(&rect1, &rect2)

| | | |
|---|---|---|
| **Syntax** | **rect_Within(&rect1, &rect2);** | |
| | | |
| **Arguments** | **&rect1, &rect2** | |
| | **rect1** | Specifies the coordinates of rectangle 1. |
| | **rect2** | Specifies the coordinates of rectangle 2. |
| | The arguments should be an array of 4 words. | |
| | | |
| **Returns** | **Status** | |
| | Status | **1:** True<br>**0:** False. |
| | | |
| **Description** | Return true if rect1 is fully within rect2. Each rectangle is an array of 4 words in the format.<br>element 0 = RECT_LEFT<br>element 1 = RECT_TOP<br>element 2 = RECT_RIGHT<br>element 3 = RECT_BOTTOM | |
| | | |
| **Example** | `rect_Intersect(box1, box2);` | |

### 2.6.31 gfx_Set(function, value)

| | | |
|---|---|---|
| **Syntax** | **gfx_Set(function, value);** | |
| | | |
| **Arguments** | **function, value** | |
| | **function** | The function number determines the required action for various graphics control functions. Usually a constant, but can be a variable, array element, or expression. There are pre-defined constants for each of the functions. |
| | **value** | A variable, array element, expression or constant holding a value for the selected function. |
| | | |
| **Returns** | **value** | |
| | **value** | Returns Previous value before change is made |
| | | |
| **Description** | Given a function number and a value, set the required graphics control parameter, such as size, colour, and other parameters. (see the **Single parameter short-cuts for the gfx_Set functions** below). | |

| function | | | value |
|---|---|---|---|
| **#** | **Predefined Name** | **Description** | |
| 0 | PEN_SIZE | Set the draw mode for gfx_LineTo, gfx_LineRel, gfx_Dot, gfx_Bullet and gfx_BoxTo (default mode is OUTLINE)<br>nb:- pen size is set to OUTLINE for normal operation | 0 or SOLID<br>1 or OUTLINE |
| 1 | BACKGROUND_COLOUR | Set the screen background colour | Colour, 0-65535 |
| 2 | OBJECT_COLOUR | Generic colour for gfx_LineTo(...), gfx_LineRel(...), gfx_Dot(), gfx_Bullet(...) and gfx_BoxTo(...) | Colour, 0-65535 |
| 3 | CLIPPING | Turns clipping on/off.<br>The clipping points are set with **gfx_ClipWindow(...)** | 0 or 1 (ON or OFF) |
| 4 | TRANSPARENT_COLOUR | Sets Bitmap, Image or Animation Transparency Colour. Defines the colour in a bitmap that inhibits writing of that colour. | Colour 0-65535 Black to White |
| 5 | TRANSPARENCY | Enables/Disables the Transparency feature**.**<br>**ENABLE:** All pixels written,<br>**DISABLE:** Pixels of TRANSPARENT_COLOUR are not written. | 0 ENABLE<br>1 DISABLE |
| 6 | FRAME_DELAY | Set the inter frame delay for **media_Video(...).**<br>This setting will over-ride the embedded frame delay of the clip. After the event, the setting will auto-disable, and if further inter-frame delays need overriding the setting must be reissued.<br>This function will not control frame delays for a image control, refer to image control. | 0 to 255msec |
| 7 | SCREEN_MODE | Set the orientation of the screen. | NORTH |

| | | | |
|---|---|---|---|
| | | NORTH or **LANDSCAPE**<br>SOUTH or **LANDSCAPE_R**<br>WEST or **PORTRAIT**<br>EAST or **PORTRAIT_R** | **SOUTH**<br>**WEST**<br>**EAST**<br>**NORTH_MIRRORED**<br>**SOUTH_MIRRORED**<br>**WEST_MIRRORED**<br>**EAST_MIRRORED** |
| 8 | **OUTLINE_COLOUR** | Outline colour for rectangles and circles<br>(set to 0 for no effect) | Colour, 0-65535 |
| 9 | **CONTRAST** | Set contrast value, 0 = display off, 1-16 = contrast level | 0 or **OFF**<br>1 to 16 for levels |
| 10 | **LINE_PATTERN** | Sets the line draw pattern for line drawing. If set to zero, lines are solid, else each '1' bit represents a pixel that is turned off. See code examples for further reference. | 0 bits for pixels on<br>1 bits for pixels off |

Single parameter short-cuts for the gfx_Set(..) functions

| **Function Syntax** | **Function Action** | **value** |
|---|---|---|
| **gfx_PenSize(mode)** | Set the draw mode for gfx_LineTo, gfx_LineRel, gfx_Dot, gfx_Bullet and gfx_BoxTo<br>**Note**: pen size is set to **OUTLINE** for normal operation (default). | 0 or **SOLID**<br>1 or **OUTLINE** |
| **gfx_BGcolour(colour)** | Set the screen  background colour | Colour 0-65535 |
| **gfx_ObjectColour(colour)** | Generic colour for gfx_LineTo(...), gfx_LineRel(...), gfx_Dot(), gfx_Bullet(... and gfx_BoxTo | Colour 0-65535 |
| **gfx_Clipping(mode)** | Turns clipping on/off.<br>The clipping points are set with **gfx_ClipWindow(...)** | 0 or 1 (**ON** or **OFF**) |
| **gfx_TransparentColour(colour)** | Sets Bitmap, Image or Animation Transparency Colour. | Colour 0-65535<br>Black to White |
| **gfx_Transparency(mode)** | Enables/Disables the Transparency feature**.** | 0 **ENABLE**<br>1  **DISABLE** |
| **gfx_FrameDelay(delay)** | Set the inter frame delay for **media_Video(...)** | 0 to 255msec |
| **gfx_ScreenMode(mode)** | Set the orientation of the screen.<br>NORTH or **LANDSCAPE**<br>SOUTH or **LANDSCAPE_R**<br>WEST or **PORTRAIT**<br>EAST or **PORTRAIT_R** | **NORTH**<br>**SOUTH**<br>**WEST**<br>**EAST**<br>**NORTH_MIRRORED**<br>**SOUTH_MIRRORED**<br>**WEST_MIRRORED**<br>**EAST_MIRRORED** |
| **gfx_OutlineColour(colour)** | Outline colour for rectangles and circles.<br>(set to 0 for no effect) | Colour 0-65535 |
| **gfx_Contrast(value)** | Set contrast value, 0 = display off, 1-16 = contrast level. | 0 or **OFF**<br>1 to 16 for levels |
| **gfx_LinePattern(pattern)** | Sets the line draw pattern for line drawing. If set to zero, lines are solid, else eac '1' bit represents a pixel that is turned off. See code examples for further reference. | 0 bits for pixels on<br>1 bits for pixels off |
| **Note:** All the shortcut commands return Previous value before change is made | | |

## 2.7 Display I/O Functions

These functions allow direct display access for fast blitting operations.

**Summary of Functions in this section:**

- disp_setGRAM(x1, y1, x2, y2)
- disp_WriteControl(value)
- disp_WriteByte(value)
- disp_WrGRAM(value)
- disp_ReadByte()
- disp_RdGRAM()
- disp_BlitPixelFill(colour, count)
- disp_BlitPixelsToMedia()
- disp_BlitPixelsFromMedia(pixelcount)
- disp_SkipPixelsFromMedia(pixelcount)
- disp_BlitPixelsToArray(dest, count)
- disp_BlitPixelsFromArray(source, count)
- disp_Scroll(x1, y1, x2, y2, mode, lines, bufptr)

### 2.7.1 disp_setGRAM(x1, y1, x2, y2)

| | | |
|---|---|---|
| **Syntax** | **disp_setGRAM(x1, y1, x2, y2);** | |
| | | |
| **Arguments** | **x1, y1, x2, y2** | |
| | **x1, y1** | Top left of the rectangular region to be selected. |
| | **x2, y2** | Bottom right of the rectangular to be selected. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **pixel_count** | |
| | **pixel_count** | The pixel count of the selected area. |
| | | |
| **Description** | Sets the hardware GRAM registers to a rectangular area, ready for writing.<br>The function returns the pixel count of the selected region, this count may then be used as an iterator for the loop that writes (whatever) to the selected GRAM area. disp_setGRAM works independantly from the clip region and does not disturb the clip area values. | |
| | | |
| **Example** | `n := disp_setGRAM(10, 10, 40, 40)` | |

### 2.7.2 disp_WriteControl(value)

| | | |
|---|---|---|
| **Syntax** | **disp_WriteControl(value);** | |
| | | |
| **Arguments** | **value** | |
| | **value** | Specifies the value to be written to the display control register. Only the lower 8 bits are sent to the display. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Sends a single byte (which is the lower 8 bits of ***value)*** to the display bus. Refer to individual data sheets for the display for more information. This function is used to extend the capabilities of the user code to gain access to the the display hardware. | |
| | | |

### 2.7.3 disp_WriteByte(value)

| | | |
|---|---|---|
| **Syntax** | **disp_WriteByte(value);** | |
| | | |
| **Arguments** | **value** | |
| | **value** | Specifies the value to be written to the display data register. Only the lower 8 bits are sent to the display. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Sends a single byte (which is the lower 8 bits of ***value)*** to the display bus. Refer to individual data sheets for the display for more information. This function is used to extend the capabilities of the user code to gain access to the the display hardware. | |
| | | |

### 2.7.4 disp_WrGRAM(value)

| | | |
|---|---|---|
| **Syntax** | **disp_WrGRAM(value);** | |
| | | |
| **Arguments** | **value** | |
| | **value** | Specifies the color value to be written to the display region selected. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Write a 16bit word to the display after an internal register or GRAM access has been set. | |
| | | |
| **Example** | `gfx_ClipWindow(40,40,44,44);   // within a small block on the display`<br>`gfx_FocusWindow();              // focus GRAM`<br><br>//disp_setGRAM can be used to set the area.<br>`disp_WrGRAM(BLUE);` | |

### 2.7.5 disp_ReadByte()

| | | |
|---|---|---|
| **Syntax** | **disp_ReadByte();** | |
| **Arguments** | **none** | |
| **Returns** | **value** | |
| | **value** | Returns the 8bit data that was read from the display. Only the lower 8bits are valid. |
| **Description** | Reads a byte from the display after an internal register or GRAM access has been set. | |
| **Example** | `gfx_ClipWindow(40,40,44,44); // within a small block on the display`<br>`gfx_FocusWindow();            // focus GRAM`<br>`pixel_Hi:= dispReadByte();    // read hi byte of first pixel`<br>`pixel_Lo:= dispReadByte();    // read lo byte of first pixel` | |

### 2.7.6 disp_RdGRAM()

| | | |
|---|---|---|
| **Syntax** | **disp_RdGRAM();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **value** | |
| | **value** | 16 bit pixel colour  HI:LO order |
| | | |
| **Description** | Reads a 16bit word from the display at the current GRAM position.<br>disp_setGRAM is usually used to set the rectangular area. Subsequent calls to disp_RdGRAM() will return consecutive pixels from the GRAM area. | |
| | | |
| **Example** | `pixel := RdGRAM(); // read pixel from GRAM, HI:LO order` | |

### 2.7.7 disp_BlitPixelFill(colour, count)

| **Syntax** | **disp_BlitPixelFill(colour, count);** | |
|---|---|---|
| | | |
| **Arguments** | **colour, count** | |
| | **colour** | Specifies the colour for the fill. |
| | **count** | Specifies the number of pixels to fill. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Fills a preselected GRAM screen area with the specified colour. | |
| | | |
| **Example** | `gfx_ClipWindow(40,40,79,79);        // select a block on the display`<br>`count := gfx_FocusWindow();         // focus GRAM`<br>`myvar:=dispBlitPixelFill(RED,count); // paint the area red` | |

## 2.7.8 disp_BlitPixelsToMedia()

| | | |
|---|---|---|
| **Syntax** | **disp_BlitPixelsToMedia();** | |
| **Arguments** | **none** | |
| **Returns** | **pixelcount** | |
| | **pixelcount** | Returns the number of pixels that were written to the media. |
| **Description** | Write the selected GRAM area to the media at the current media address. | |

**Example**

```
func main()
    var n;
    while(!media_Init())
        putstr("Insert Card");         // init the card
        pause(200);
        gfx_Cls();
        pause(200);
    wend
    media_SetSector(0x0020,0x0000);   // we're going to write here
    gfx_ClipWindow(40,40,55,55); // select 16x16 block on the display
    n:=gfx_FocusWindow();             // focus GRAM
    while(n--)
        disp_BlitPixelFill(RAND(),1); // fill area with random pixels
    wend
    n:=disp_BlitPixelsToMedia ();     // save it to sector
    print(n*2," bytes written\n");
    print("Done!");
    repeat forever
endfunc
```

### 2.7.9 disp_BlitPixelsFromMedia(pixelcount)

| | | |
|---|---|---|
| **Syntax** | **disp_BlitPixelFromMedia(pixelcount);** | |
| | | |
| **Arguments** | **pixelcount** | |
| | **pixelcount** | Specifying the number of pixels to be consecutively read from the media stream. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Read the required number of pixels consecutively from the current media stream and write them to the current display GRAM address. For 8bit colour mode, each pixel comprises a single 8bit value. For 16bit colour, each pixel is composed of 2 bytes, the high order byte is read first, the low order bye is read next. | |
| | | |
| **Example** | `...`<br>`media_SetAdd(0x0002, 0x3C00);    // point to required area of an image`<br>`disp_BlitPixelsFromMedia(20);    // write the next 20 pixels from`<br>`                                 // media to the current GRAM pointer.`<br>`...` | |

### 2.7.10 disp_SkipPixelsFromMedia(pixelcount)

| | | |
|---|---|---|
| **Syntax** | **disp_BlitPixelFromMedia(pixelcount);** | |
| | | |
| **Arguments** | **pixelcount** | |
| | **pixelcount** | Specifying the number of pixels to be consecutively skipped from the media stream. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Skip the required number of pixels consecutively from the current media stream, discarding them. For 8bit colour mode, each pixel comprises a single 8bit value. For 16bit colour, each pixel is composed of 2 bytes, the high order byte is read first, the low order bye is read next. | |
| | | |

**Example**

```
...
disp_SkipPixelsFromMedia(20);    // skip the next 20 pixels from media
disp_BlitPixelsFromMedia(20);    // write the next 20 pixels from
                                 // media to the current GRAM pointer.
...
```

### 2.7.11 disp_BlitPixelsToArray(dest, count)

| | | |
|---|---|---|
| **Syntax** | **disp_BlitPixelsToArray(dest, count);** | |
| | | |
| **Arguments** | **dest, count** | |
| | **dest** | Buffer to store pixels(colour value) from the GRAM. |
| | **count** | Number of pixels to be written to the buffer. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Reads "count" pixels (words) from the GRAM to a buffer. After the read the GRAM pointer will have been incremented by "count". | |
| | | |
| **Example** | `disp_BlitPixelsToArray(buffer, 20);` | |

## 2.7.12 disp_BlitPixelsFromArray(source, count)

| | | |
|---|---|---|
| **Syntax** | **disp_BlitPixelsFromArray(source, count);** | |
| | | |
| **Arguments** | **source, count** | |
| | **source** | Buffer with pixels(colour value) to be copied to the GRAM. |
| | **count** | Number of pixels to be written to the GRAM. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Stores "count" pixels to the GRAM from a buffer.<br>After the write the GRAM pointer will have been incremented by "count". | |
| | | |
| **Example** | `disp_BlitPixelsFromArray(buffer, 20);` | |

## 2.7.13 disp_Scroll(x1, y1, x2, y2, mode, lines, bufptr)

| | | |
|---|---|---|
| **Syntax** | **disp_Scroll(x1, y1, x2, y2, mode, lines, bufptr);** | |
| | | |
| **Arguments** | **x1, y1, x2, y2, mode, lines, bufptr** | |
| | **x1, y1** | Top left of the rectangular area to be scrolled. |
| | **x2, y2** | Bottom right of the rectangular area to be scrolled. |
| | **mode** | Scroll Direction<br>'UP'<br>'DOWN'<br>'LEFT'<br>'RIGHT' |
| | **lines** | Number of lines to be scrolled at a time |
| | **bufptr** | An array for holding intermediate pixel transfer data. The array must be x2-x1+1 words for vertical scrolling, and y2-y1+1 words for horizontal scrolling. If the buffer is not large enough, the variables above the buffer will be corrupted - there is no checking and it is up to the caller to make sure the buffer is large enough.<br><br>On returning to the caller, the buffer will still contain the pixel values of last line that was moved. |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Scroll vertically or horizontally within an area defined by x1,y1,x2,y2 in the direction specified by "mode". | |
| | | |
| **Example** | `disp_Scroll(10,10,118,118,DOWN,3, buffer);`<br>`//scroll area downwards 3 lines at a time;` | |

## 2.8 Media Functions (SD/SDHC Memory Card or Serial Flash chip)

The media can be SD/SDHC, microSD or serial (NAND) flash device interfaced to the GOLDELOX-PoGa SPI port.

**Summary of Functions in this section:**

- media_Init()
- media_SetAdd(HIword, LOword)
- media_SetSector(HIword, LOword)
- media_ReadByte()
- media_ReadWord()
- media_WriteByte(byte_val)
- media_WriteWord(word_val)
- media_Flush()
- media_Image(x, y)
- media_Video(x, y)
- media_VideoFrame(x, y, frameNumber)
- media_SelectGCIimage(entrynum, frame,mode)
- media_Offset(sector)
- media_LoadArray(dest, count)
- media_StoreArray(source, count)
- media_LoadImageHeader()
- media_SetScanLine(line, offset)
- media_PoGaFile(filenumber)

### 2.8.1 media_Init()

| | | |
|---|---|---|
| **Syntax** | **media_Init();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **result** | |
| | **result** | Returns: **1** if memory card is present and successfully initialised<br>Returns: **0** if no card is present or not able to initialise |
| | | |
| **Description** | Initialise a uSD/SD/SDHC memory card for further operations. The SD card is connected to the SPI (serial peripheral interface) of the GOLDELOX-PoGa chip. | |
| | | |

**Example**

```
while(!media_Init())
    gfx_Cls();
    pause(300);
    puts(“Please insert SD card”);
    pause(300);
wend
```

This example waits for SD card to be inserted and initialised, flashing a message if no SD card detected.

## 2.8.2 media_SetAdd(HIword, LOword)

| | | |
|---|---|---|
| **Syntax** | **media_SetAdd(HIword, LOword);** | |
| | | |
| **Arguments** | **HIword, LOword** | |
| | **HIword** | specifies the high word (upper 2 bytes) of a 4 byte media memory byte address location. |
| | **LOword** | specifies the low word (lower 2 bytes) of a 4 byte media memory byte address location. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Set media memory internal Address pointer for access at a non sector aligned byte address. | |
| | | |
| **Example** | `media_SetAdd(0, 513);`<br>This example sets the media address to byte 513 (which is sector #1, 2nd byte in sector) for subsequent operations. | |

### 2.8.3 media_SetSector(HIword, LOword)

| | | |
|---|---|---|
| **Syntax** | **media_SetSector(HIword, LOword);** | |
| | | |
| **Arguments** | **HIword, LOword** | |
| | **HIword** | specifies the high word (upper 2 bytes) of a 4 byte media memory sector address location. |
| | **LOword** | specifies the low word (lower 2 bytes) of a 4 byte media memory sector address location. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Set media memory internal Address pointer for sector access. | |
| | | |
| **Example** | `media_SetSector(0, 10);`<br>This example sets the media address to the 11th sector (which is also byte address 5120) for subsequent operations | |

### 2.8.4 media_ReadByte()

| **Syntax** | **media_ReadByte();** |
|---|---|
| | |
| **Arguments** | **none** |
| | |
| **Returns** | **byte value** |
| | |
| **Description** | Returns the byte value from the current media address. The internal byte address will then be internally incremented by one. |
| | |
| **Example** | |

```
var LObyte, HIbyte;
if(media_Init())
   media_SetAdd(0, 510);
   LObyte := media_ReadByte();
   HIbyte := media_ReadByte();
   print([HEX2]HIbyte,[HEX2]LObyte);
endif
repeat forever
```

This example initialises the media, sets the media byte address to 510, and reads the last 2 bytes from sector 0. If the card happens to be FAT formatted, the result will be “AA55”. The media internal address is internally incremented for each of the byte operations.

## 2.8.5 media_ReadWord()

| | |
|---|---|
| **Syntax** | **media_ReadWord();** |
| | |
| **Arguments** | **none** |
| | |
| **Returns** | **word value** |
| | |
| **Description** | Returns the word value (2 bytes) from the current media address. The internal byte address will then be internally incremented by one. If the address is not aligned, the word will still be read correctly. |
| | |
| **Example** | `var myword;`<br>`if(media_Init())`<br>`   media_SetAdd(0, 510);`<br>`   myword := media_ReadWord();`<br>`   print([HEX4]myword);`<br>`endif`<br>`repeat forever`<br><br>This example initialises the media, sets the media byte address to 510 and reads the last word from sector 0. If the card happens to be formatted, the result will be “AA55” |

### 2.8.6 media_WriteByte(byte_val)

| | | |
|---|---|---|
| **Syntax** | **media_WriteByte(byte_val);** | |
| | | |
| **Arguments** | **byte_val** | |
| | **byte_val** | The lower 8 bits specifies the byte to be written at the current media address location. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **success** | |
| | **success** | Returns non zero if write was successful. |
| | | |
| **Description** | Writes a byte to the current media address that was initially set with **media_SetSector(...);**<br>**Note:** Due to design constraints on the GOLDELOX-PoGa, there is no way of writing bytes or words within a media sector without starting from the beginning of the sector. All writes will start at the beginning of a sector and are incremental until the **media_Flush()** function is executed, or the sector address rolls over to the next sector. Any remaining bytes in the sector will be padded with **0xFF**, destroying the previous contents. An attempt to use the **media_SetAdd(..)** function will result in the lower 9 bits being interpreted as zero. If the writing rolls over to the next sector, the media_Flush() function is issued automatically internally. | |
| | | |

**Example**

```
var n, char;
while (media_Init()==0);    // wait if no SD card detected
media_SetSector(0, 2);      // at sector 2
//media_SetAdd(0, 1024);    // (alternatively, use media_SetAdd(),
                            // lower 9 bits ignored)
while (n < 10)
    media_WriteByte(n++ +'0');  // write ASCII '0123456789' to the
wend                            // first 10 locations.

to(MDA); putstr("Hello World"); // now write a ascii test string
media_WriteByte('A');           // write a further 3 bytes
media_WriteByte('B');
media_WriteByte('C');
media_WriteByte(0);             // terminate with zero
media_Flush();                  // we're finished, close the sector

media_SetAdd(0, 1024+5);        // set the starting byte address
while(char:=media_ReadByte()) putch(char); // print result, starting
                                           // from '5'
repeat forever
```

This example initialises the media, writes some bytes to the required sector, then prints the result from the required location.

### 2.8.7 media_WriteWord(word_val)

| | | |
|---|---|---|
| **Syntax** | **media_WriteWord(word_val);** | |
| | | |
| **Arguments** | **word_val** | |
| | **word_val** | The 16 bit word to be written at the current media address location. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **success** | |
| | **success** | Returns non zero if write was successful. |
| | | |
| **Description** | Writes a byte to the current media address that was initially set with **media_SetSector(...);**<br>**Note:** Due to design constraints on the GOLDELOX-PoGa, there is no way of writing bytes or words within a media sector without starting from the beginning of the sector. All writes will start at the beginning of a sector and are incremental until the **media_Flush()** function is executed, or the sector address rolls over to the next sector. Any remaining bytes in the sector will be padded with **0xFF**, destroying the previous contents. An attempt to use the **media_SetAdd(..)** function will result in the lower 9 bits being interpreted as zero. If the writing rolls over to the next sector, the media_Flush() function is issued automatically internally. | |
| | | |
| **Example** | | |

```
var n;

while (media_Init()==0);     // wait until a good SD card is found
n:=0;
media_SetAdd(0, 1536);       // set the starting byte address
while (n++ < 20)
    media_WriteWord(RAND()); // write 20 random words to first 20
wend                         // word locations.
n:=0;
while (n++ < 20)
    media_WriteWord(n++*1000);// write sequence of 1000*n to next 20
wend                          // word locations.
media_Flush();                // we're finished, close the sector

media_SetAdd(0, 1536+40);     // set the starting byte address
n:=0;
while(n++<8)         // print result of fist 8 multiplication calcs
    print([HEX4] media_ReadWord(),"\n");
wend
repeat forever
```

This example initialises the media, writes some words to the required sector, then prints the result from the required location.

### 2.8.8 media_Flush()

| **Syntax** | **media_Flush();** |
|---|---|
| | |
| **Arguments** | **none** |
| | |
| **Returns** | **nothing** |
| | |
| **Description** | After writing any data to a sector, media_Flush() should be called to ensure that the current sector that is being written is correctly stored back to the media else write operations may be unpredictable. |
| | |
| **Example** | See the media_WriteByte(..) and media_WriteWord(..) examples. |

### 2.8.9 media_Image(x, y)

| | | |
|---|---|---|
| **Syntax** | **media_Image(x, y);** | |
| | | |
| **Arguments** | **x, y** | |
| | **x, y** | specifies the top left position where the image will be displayed. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Displays an image from the media storage at the specified co-ordinates. The image address is previously specified with the **media_SetAdd(..)** or **media_SetSector(...) function**. If the image is shown partially off screen, it is necessary to enable clipping for it to be displayed correctly.<br>**Note:** it is assumed that the media has been loaded with the example images in GFX2DEMO.GCI loaded at sector 0. This can be loaded using the Graphics Composer (directly onto the memory card. | |
| | | |

**Example**

```
while(media_Init()==0);          // wait if no SD card detected

media_SetAdd(0x0001, 0xDA00);    // point to the books04 image
media_Image(10,10);
gfx_Clipping(ON);        // turn off clipping to see the difference
media_Image(-12,50);     // show image off-screen to the left
media_Image(50,-12);     // show image off-screen at the top
repeat forever
```

This example draws an image at several positions, showing the effects of clipping.

### 2.8.10 media_Video(x, y)

| | | |
|---|---|---|
| **Syntax** | **media_Video(x, y);** | |
| | | |
| **Arguments** | **x, y** | |
| | **x, y** | specifies the top left position where the video clip will be displayed. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Displays a *video* clip from the media storage device at the specified co-ordinates. The *video* address location in the media is previously specified with the **media_SetAdd(..)** or **media_SetSector(...)** function. If the *video* is shown partially off screen, it is necessary to enable clipping for it be displayed correctly. Note that showing a *video* blocks all other processes until the video has finished showing. See the **media_VideoFrame(...)** functions for alternatives.<br><br>**Note:** it is assumed that the media has been loaded with the example video in GFX2DEMO.GCI loaded at sector 0. This can be loaded using the Graphics Composer directly onto the memory card. | |
| | | |

**Example**

```
while(media_Init()==0);         // wait if no SD card detected

media_SetAdd(0x0001, 0x3C00);   // point to the 10-gear clip
media_Video(10,10);
gfx_Clipping(ON);      // turn off clipping to see the difference
media_Video(-12,50);   // show video off-screen to the left
media_Video(50,-12);   // show video off-screen at the top
repeat forever

```

This example plays a video clip at several positions, showing the effects of clipping.

### 2.8.11 media_VideoFrame(x, y, frameNumber)

| | | |
|---|---|---|
| **Syntax** | **media_VideoFrame(x, y, frameNumber);** | |
| **Arguments** | **x, y** | |
| | **x, y** | specifies the top left position where the video clip will be displayed. |
| | **frameNumber** | Specifies the required frame to be shown. |
| | The arguments can be a variable, array element, expression or constant | |
| **Returns** | **nothing** | |

**Description**

Displays a *video* from the media storage device at the specified co-ordinates. The *video* address is previously specified with the **media_SetAdd(..)** or **media_SetSector(...)** function. If the *video* is shown partially off screen, it is necessary to enable clipping for it be displayed correctly. The frames can be shown in any order. This function gives you great flexibility for showing various icons from an image strip, as well as showing videos while doing other tasks

**Note:** it is assumed that the media has been loaded with the example video in GFX2DEMO.GCI loaded at sector 0. This can be loaded using the Graphics Composer directly onto the memory card.

**Example**

```
var frame;
while (media_Init()==0);  // wait if no SD card detected

while (media_Init()==0);        // wait if no SD card detected
media_SetAdd(0x0002, 0x3C00);   // point to the 10-gear image
repeat
    frame := 0;  // start at frame 0
    repeat
        media_VideoFrame(30,30, frame++); // display a frame
        pause(peekB(IMAGE_DELAY));  // pause for the time given in
                                    // the image header
    until(frame == peekW(IMG_FRAME_COUNT));  // loop until we've
                                             // shown all the frames
forever // do it forever

```

This first example shows how to display frames as required while possibly doing other tasks. Note that the frame timing (although not noticeable in this small example) is not correct as the delay commences after the image frame is shown, therefore adding the display overheads to the frame delay. This second example employs a timer for the framing delay, and shows the same movie simultaneously running forward and backwards with time left for other tasks as well. A number of videos (or animated icons) can be shown simultaneously using this method.

```
var framecount, frame, delay, colr;
frame := 0;
// show the first frame so we can get the video header info
// into the system variables, and then to our local variables.
media_VideoFrame(30,30, 0);

framecount := peekW(IMG_FRAME_COUNT);  // we can now set some local
                                       // values.
delay := peekB(IMAGE_DELAY);  // get the frame count and delay
repeat
    repeat
        pokeW(TIMER0, delay);              // set a timer
        media_VideoFrame(30,30, frame++);  // show next frame
        gfx_MoveTo(64,35);
        print([DEC2Z] frame);              // print the frame number
        media_VideoFrame(30,80, framecount-frame);   // show movie
                                                     // backwards.
        gfx_MoveTo(64,85);
        print([DEC2Z] framecount-frame);    // print the frame number

        if ((frame & 3) == 0)
            gfx_CircleFilled(80,20,2,colr);  // a blinking circle fun
            colr := colr ^ 0xF800;           // alternate colour,
        endif                                // BLACK/RED using XOR
        // do more here if required
        while(peekW(TIMER0));    // wait for timer to expire
    until(frame == peekW(IMG_FRAME_COUNT));
    frame := 0;
forever
```

### 2.8.12 media_SelectGCIimage(entrynum, frame, mode)

| | | |
|---|---|---|
| **Syntax** | **media_SelectGCIimage(entrynum, frame,mode);** | |
| | | |
| **Arguments** | **entrynum, frame, mode** | |
| | **entrynum** | GCI Entry number. |
| | **frame** | Frame number in the selected entry. |
| | **mode** | **Mode = 1**<br>Displays the required frame of the image at the preset position (determined by the xpos and ypos setting in the GIC file).<br>**Mode = 0**<br>The frame is displayed at the position that is previously set with gfx_MoveTo. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **Frame number** | |
| | **Frame number** | Returns the next frame number. |
| | | |
| **Description** | This function is used to easily display an image in a GCI entry within a PoGa file. Up to 256 images or movies are stored for use by each PoGa file entry (constrained by the maximum size allowed for the GCI data which is 3960 sectors (2,027,520 bytes) | |
| | | |
| **Example** | `media_SelectGCIimage(10, 0, 1);`<br>`// select entry #10, frame 0, use entries co-ordinates` | |

## 2.8.13 media_Offset(sector)

| | | |
|---|---|---|
| **Syntax** | **media_Offset(sector);** | |
| | | |
| **Arguments** | **sector** | |
| | **sector** | Offset to be set for the uSD internal sectors. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **Nothing** | |
| | | |
| **Description** | Set uSD internal Sector offset for sector relative block access. Some constants are already provided for access to relevant areas in the PoGa file structure.<br>These are.<br>POGA_FCB_OFFSET 0 // sector offset to PoGa File Control Block<br>GCI_MAP_OFFSET 56 // sector offset to GCI entry map<br>BIN_OFFSET 72 // sector offset to BIN dada in PoGa file<br>GCI_IMAGE_OFFSET 136 // sector offset to GCI images in PoGa file<br><br>**Note:** When the function media_PoGaFile(filenumber) is executed with a value of 1-512 to select a PoGa file base offset, the media offset for GCI_DATA_OFFSET is set, however, if media_PoGaFile(0) is selected, the offset will be set to zero. | |
| | | |
| **Example** | `media_Offset(136);` | |

### 2.8.14 media_LoadArray(dest, count)

| | | |
|---|---|---|
| **Syntax** | **media_LoadArray(dest, count);** | |
| | | |
| **Arguments** | **dest, count** | |
| | **dest** | Buffer to load the byte values to. |
| | **count** | Number of bytes to be read. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **Nothing** | |
| | | |
| **Description** | Reads "count" bytes of data from the uSD card pointed to by the internal Address pointer to a buffer. After the read the Address pointer is automatically incremented by "count". | |
| | | |
| **Example** | `media_LoadArray(buffer, 20);` | |

## 2.8.15 media_StoreArray(source, count)

| | | |
|---|---|---|
| **Syntax** | **media_StoreArray(source, count);** | |
| | | |
| **Arguments** | **source, count** | |
| | **source** | Buffer to load the byte values from. |
| | **count** | Number of bytes to be written. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **Nothing** | |
| | | |
| **Description** | Stores "count" bytes of data to the uSD card pointed to by the internal Address pointer from a buffer. After the read the Address pointer is automatically incremented by "count". | |
| | | |
| **Example** | `media_StoreArray(buffer, 20);` | |

## 2.8.16 media_LoadImageHeader()

| | | |
|---|---|---|
| **Syntax** | **media_LoadImageHeader();** | |
| | | |
| **Arguments** | **Nothing** | |
| | | |
| **Returns** | **Frame count** | |
| | **Frame count** | Returns the next frame count. |
| | | |
| **Description** | Loads the image header at the current media address.<br>The address of the image must first be specified by media_setSector function, returns zero if failed or returns frame count.<br><br>If frame count == 1, header is a single image<br><br>This function sets the following internal byte variables which can then be read with peekB:-<br>IMAGE_WIDTH 136 // (byte) width of loaded image<br>IMAGE_HEIGHT 137 // (byte) height of loaded image<br>IMAGE_DELAY 138 // (byte) 0 if image, else inter frame delay for movie<br>IMAGE_MODE 139 // (byte) bit 4 determines colour mode (not used) , other bits reserved<br>// and these then be read with peekW:-<br>IMG_PIXEL_COUNT 90 // pixel count of current object (may be altered by clipping)<br>IMG_FRAME_COUNT 91 // (word) count of frames in currently loaded video | |
| | | |
| **Example** | `var := media_LoadImageHeader(); // load image header, get frame count` | |

## 2.8.17 media_SetScanLine(line, offset)

| | | |
|---|---|---|
| **Syntax** | **media_SetScanLine(line, offset);** | |
| | | |
| **Arguments** | **line, offset** | |
| | **line** | selects the required scan line from within an image |
| | **offset** | the offset from the 'left edge' of the scanlin |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **Nothing** | |
| | | |
| **Description** | This function assumes that a sector start address for an image has been set, and a valid header has been loaded with media_LoadImageHeader. The function calculates the starting line in a vertical image strip, ready for blitting to GRAM using the disp_BlitPixelsFromMedia function. Normally, this function is used with an 'image strip' , however, the media address may also be offset from the left hand side of the image, allowing a large image to be panned in smaller sections.<br>The GRAM area must be set correctly, and the number of pixels blitted must agree with the GRAM area else the image portion will not be transferred correctly. | |
| | | |
| **Example** | media_SetScanLine(20,0);<br>// set the media address to the start of the 20th line | |

## 2.8.18 media_PoGaFile(filenumber)

| | | |
|---|---|---|
| **Syntax** | **media_PoGaFile(filenumber);** | |
| | | |
| **Arguments** | **filenumber** | |
| | **filenumber** | Given a file number 1-255, internally adds the correct base offset address for media operations. If the number is zero, there is no offsets added. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **Nothing** | |
| | | |
| **Description** | 1]<br>media_PoGaFile(0 or OFF);<br>PoGa file system is off, all media addresses are relative to offset zero, this is legacy Goldelox behaviour.<br><br>**Note:** Beware!!, writes to areas below sector 0x20000 will clobber a PoGa disk structure.<br><br>2]<br>media_PoGaFile(1 to 512);<br>PoGa file selected, writing to media is disabled, reading is relative to start of PoGa file.<br><br>**Note:** Any value >512 will be ignored. | |
| | | |
| **Example** | `See Description.` | |

## 2.9 PoGa File System Operations

**Summary of Functions in this section:**

- func RunProgram(page)
- LoadProgram(ByteCount, page)

### 2.9.1 RunProgram(page)

| | | |
|---|---|---|
| **Syntax** | **RunProgram(page);** | |
| | | |
| **Arguments** | **page** | |
| | **page** | Select the 1k page offset where the 4DGL program will be executed from.<br>PoGa has 11k of FLASH, pages 0-10 |
| | | |
| **Returns** | **result** | |
| | result | The 4DGL program will be executed from the required page address. |
| | | |
| **Description** | RunProgram is usually only used by a menu control programto start execution of an overlay above the menu control program. Control is passed to a program that is expected to be at the page specified, usually one that has been loaded from PoGa disk. If the program being run returns it will restart the menu program program at page 0, ie Control does not return to the instruction following the RunProgram function, the system does a warm boot to the menu control program at page zero.<br><br>Normally, it is desirable to keep a menu program small so it will fit in the first 1k bolck, this allows 10k free for programs loaded from PoGa disk, however, the menu program can be made larger if required, therefore it is possible to start execution of programs at say, block 2 , allowing some flexibility to accomodate a larger menu or control program. | |
| | | |
| **Example** | `RunProgram(1);  // eg run program at page 1` | |

### 2.9.2 LoadProgram(ByteCount, page)

| | | |
|---|---|---|
| **Syntax** | **LoadProgram(ByteCount, page)** | |
| | | |
| **Arguments** | **ByteCount, page** | |
| | **ByteCount** | Number of bytes to be loaded on the page. |
| | **page** | Determines the loading location for the program in 1k steps |
| | | |
| **Returns** | **checksum** | |
| | checksum | returns the checksum of the loaded program. |
| | | |
| **Description** | Loads 4DGL FLASH program area from FLASH.<br>Total program space is 11 x 1024 byte blocks, first 1024 bytes is usually reserved for a menu program. If the checksum matches the PoGa program checksum, it can then be executed with the runProgram("page"); function<br>**Note:** Refer to the pogaMenux.4DG programs for example code. | |
| | | |
| **Example** | `checksum := LoadProgram(1000, 1);`<br>`// load a 1000 byte prog from current media to page 1 of main flash` | |

## 2.10 Extended Functions

**Summary of Functions in this section:**

- iterator(offset)
- EVE_SP()

## 2.10.1 iterator(offset)

| **Syntax** | **iterator(offset);** | |
|---|---|---|
| | | |
| **Arguments** | **offset** | |
| | **offset** | The step value for the next ++ or – operation. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Mainly used for stepping a pointer through an array of elements where user has defined groups or types.<br>**Note:** INCVAL is automatically reset after the next occurrence of a ++ or -- operator. | |
| | | |
| **Example** | `iterator(arg);`<br>`// set the iterator size for ++/-- (same as pokeW(INCVAL, n);` | |

## 2.10.2 EVE_SP()

| | | |
|---|---|---|
| **Syntax** | **EVE_SP()** | |
| | | |
| **Arguments** | **Nothing** | |
| | | |
| **Returns** | **value** | |
| | value | Value expressing current status of the Stack pointer. |
| | | |
| **Description** | Spot check the current EVE stack ptr.<br>Used for debugging to keep an eye on the stack. Stack is from 512 to 720, EVE will be unstable if stack goes above 720. | |
| | | |
| **Example** | `print(EVE_SP);` | |

## 2.11 Serial (UART) Communications Functions

**Summary of Functions in this section:**

- serin()
- serout(char)
- setbaud(rate)
- com_Reset()
- com_Count()
- com_Full()
- com_Error()
- com_Sync()
- com_TX(buf, bufsize)
- com_TX_Count()
- com_CSUM_8(buf, count)
- com_CRC_16(buf, count)
- com_CRC_MODBUS(buf, count)
- com_CRC_CCITT(buf, count, seed)
- sys_EventsPostpone()
- sys_EventsResume()

### 2.11.1 serin()

| | | |
|---|---|---|
| **Syntax** | **serin();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **char** | |
| | **char** | Returns: **-1** if no character is available<br>Returns: **-2** if a framing error or over-run has occurred (auto cleared)<br>Returns: positive value **0 to 255** for a valid character received |
| | | |
| **Description** | Receives a character from the Serial Port COM0. The transmission format is:<br>**No Parity, 1 Stop Bit, 8 Data Bits** (N,8,1).<br>The default Baud Rate is 115,200 bits per second or 115,200 baud. The baud rate can be changed under program control by using the **setbaud(...)** function. | |
| | | |
| **Example** | `var char;`<br>`char := serin();     // test the com port`<br>`if (char >= 0)       // if a valid character is received`<br>`   process(char);    // process the character`<br>`endif` | |

## 2.11.2 serout(char)

| | | |
|---|---|---|
| **Syntax** | **serout(char);** | |
| | | |
| **Arguments** | **char** | |
| | **char** | specifies the data byte to be sent to the serial port. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Transmits a single byte from the Serial Port COM0. The transmission format is:<br>**No Parity, 1 Stop Bit, 8 Data Bits** (N,8,1).<br>The default Baud Rate is 115,200 bits per second or 115,200 baud. The baud rate can be changed under program control by using the **setbaud(...)** function. | |
| | | |

## 2.11.3 setbaud(rate)

| **Syntax** | **setbaud(rate);** | |
|---|---|---|
| | | |
| **Arguments** | **rate** | |
| | **rate** | Specifies the baud rate divisor value or pre-defined constant. |
| | The argument can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Use this function to set the required baud rate. The default baud rate is 115,200 baud.<br>There are pre-defined baud rate constants for most common baud rates: | |

| **Pre Defined Constant** | **Rate Divisor** | **Error %** | **Actual Baud Rate** |
|---|---|---|---|
| BAUD_110 | 27272 | 0.00% | 110 |
| BAUD_300 | 9999 | 0.00% | 300 |
| BAUD_600 | 4999 | 0.00% | 600 |
| BAUD_1200 | 2499 | 0.00% | 1200 |
| BAUD_2400 | 1249 | 0.00% | 2400 |
| BAUD_4800 | 624 | 0.00% | 4800 |
| BAUD_9600 | 312 | -0.16% | 9584 |
| BAUD_14400 | 207 | 0.16% | 14423 |
| BAUD_19200 | 155 | 0.16% | 19230 |
| BAUD_31250 | 95 | 0.00% | 31250 |
| MIDI | 95 | 0.00% | 31250 |
| BAUD_38400 | 77 | 0.16% | 38461 |
| BAUD_56000 | 53 | -0.79% | 55555 |
| BAUD_57600 | 51 | 0.16% | 57692 |
| BAUD_115200 | 25 | 0.16% | 115384 |
| BAUD_128000 | 22 | 1.90% | 130434 |
| BAUD_256000 | 11 | -2.34% | 250000 |
| BAUD_300000 | 10 | 0.00% | 300000 |
| BAUD_375000 | 8 | 0.00% | 375000 |
| BAUD_500000 | 6 | 0.00% | 500000 |
| BAUD_600000 | 4 | 0.00% | 600000 |

The baud rate is calculated with the following formula:

**rate-divisor = (3000000 / baud ) - 1**

### 2.11.4 com_Init(buffer, bufsize, serviceFunc , timeout, qualifier)

| | | |
|---|---|---|
| **Syntax** | **com_Init(buffer, bufsize, qualifier, serviceFunc, timeout, qualifier );** | |
| | | |
| **Arguments** | **buffer, bufsize, qualifier, serviceFunc, timeout, qualifier** | |
| | **buffer** | specifies the address of a buffer used for the background buffering service. |
| | **bufsize** | specifies the byte size of the user array provided for the buffer (each array element holds 2 bytes). If the buffer size is zero, a buffer of 128 words (256 bytes) should be provided for automatic packet length mode (see below). |
| | **serviceFunc** | Specify a function name to handle timeout or buffer full events. A zero indicates no event to be used. |
| | **timeout** | A value in milliseconds to set the packet timeout value. A zero indicates no timeout to be used. |
| | **qualifier** | specifies the qualifying character that must be received to initiate serial data reception and buffer write. A zero indicates no qualifier to be used. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | This is the initialisation function for the serial communications buffered service. Once initialised, the service runs in the background capturing and buffering serial data without the user application having to constantly poll the serial port. This frees up the application to service other tasks.<br>MODES OF OPERATION<br>• No *qualifier* – simple ring buffer (aka circular queue)<br>**eg com_Init(mybuffer, 64 , 0, 0, 0);**<br>If the ***qualifier, function,*** *and* ***timeout*** arguments are set to zero, the ***buffer*** is continually active in the background as a simple circular queue. Characters when received from the host are placed in the circular queue (at the 'head' of the queue) Bytes may be removed from the circular queue (from the 'tail' of the queue) using the **serin()** function. If the tail is the same position as the head, there are no bytes in the queue, therefore **serin()** will return **-1**, meaning no character is available, also, the **com_Count()** function can be read at any time to determine the number of characters that are waiting between the tail and head of the queue. If the queue is not read frequently by the application, and characters are still being sent by the host, the head will eventually catch up with the tail setting the internal COM_FULL flag (which can be read with the **com_Full()** function) . Any further characters from the host are are now discarded, however, all the characters that were buffered up to this point are readable. This is a good way of reading a fixed size packet and not necessarily considered to be an error condition (by utilising the ***function*** argument, this condition can activate a function designed to deal with the packet that has been received). If no characters are removed from the buffer until the COM_FULL flag (which can be read with the **com_Full()** function) becomes set, it is guaranteed that | |

the bytes will be ordered in the ***buffer*** from the start position, therefore, the ***buffer*** can be treated as an array and can be read directly without using **serin()** at all. In the latter case, the correct action is to process the data from the buffer, re-initialise the buffer with the **com_Init(..)** function, or reset the buffered serial service by issuing the **com_Reset()** function (which will return serial reception to polled mode) , and send an acknowledgement to the host (traditionally a **ACK** or 6) to indicate that the application is ready to receive more data and the previous 'packet' has been dealt with, or conversely, the application may send a negative acknowledgement to indicate that some sort of error occurred, or the action could not be completed (traditionally a N**AK** or 16) .

If any low level errors occur during the buffering service (such as framing or over-run) the internal COM_ERROR flag will be set (which can be read with the **com_Error()** function). Note that the COM_FULL flag will remain latched to indicate that the buffer did become full, and is not reset (even if all the characters are read) until the **com_Init(..) or com_Reset()** function is issued.

- **Using a qualifier**

  **eg com_Init(mybuffer, 64, 0, 0, ':');**

  If a ***qualifier*** character is specified, after the buffer is initialised with **com_Init(..) ,** the service will ignore all characters until the ***qualifier*** is received and only then initiate the buffer write sequence with incoming data. After that point, the behaviour is the same as above for the 'non qualified' mode. This is particularly useful for situations like MODBUS-ASCII which always has a lead-in character ':'

- **Using a timeout value**

  **eg com_Init(mybuffer, 64, serviceFunc, 1000, ':');**

  If a timeout value is used, it is usual that there is also a named function to deal with the timeout situation, in which case, the function will serve 2 purposes, that being A] The maximum amount of characters has been received, of B] The packet time has been exceeded. NB: If a timeout value is specified , TIMER3 is employed as a timeout timer. The timer will be started (with the time value specified by the ***timeout*** argument) when the first character is received, (or in the case where a qualifier is used, the timer will only be started when the qualifier is first received) Subsequent characters received will keep restarting the timer, keeping it 'topped up', but as soon as the timer times out (or the maximum number of characters that the buffer can hold has been received), the **serviceFunc** will be invoked. If **serviceFunc** is zero, the timer action will still occur, allowing the timer to be polled in a simple fashion, however there will be no event service.

### 2.11.5 com_Reset()

| | |
|---|---|
| **Syntax** | **com_Reset();** |
| | |
| **Arguments** | **none** |
| | |
| **Returns** | **nothing** |
| | |
| **Description** | Resets the serial communications buffered service and returns it to the default polled mode. |
| | |
| **Example** | `com_Reset(); // reset to polled mode` |

### 2.11.6 com_Count()

| | | |
|---|---|---|
| **Syntax** | **com_Count();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **count** | |
| | **count** | current count of  characters in the communications  buffer. |
| | | |
| **Description** | Can be read at any time (when in buffered communications is active) to determine the number of characters that are waiting in the  buffer. | |
| | | |
| **Example** | `n := com_Count(); // get the number of chars available in the buffer` | |

## 2.11.7 com_Full()

| | | |
|---|---|---|
| **Syntax** | **com_Full();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **status** | |
| | **status** | Returns **1** if buffer or queue has become full, or is overflowed, else returns **0**. |
| | | |
| **Description** | If the queue is not read frequently by the application, and characters are still being sent by the host, the head will eventually catch up with the tail setting the COM_FULL flag which is read with this function. If this flag is set, any further characters from the host are discarded, however, all the characters that were buffered up to this point are readable.<br>**Note:** If com_Init(...) function is utilizing the serviceFunc argument, the serviceFunc will have been activated. Using com_Full() within serviceFunc is the way to differentiate between a timeout activation and a com_Full activation. | |
| | | |
| **Example** | `if(com_Full() & (com_Count() == 0))`<br>`   com_Init(mybuf, 30, 0);  // buffer full, recovery`<br>`endif` | |

## 2.11.8 com_Error()

| | | |
|---|---|---|
| **Syntax** | **com_Error();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **status** | |
| | **status** | Returns **1** if any low level communications error occurred, else returns **0.** |
| | | |
| **Description** | If any low level errors occur during the buffering service (such as framing or over-run) the internal **COM_ERROR** flag will be set which can be read with this function. | |
| | | |
| **Example** | `if(com_Error())        // if there were low level comms errors,`<br>`    resetMySystem();   // take corrective action`<br>`endif` | |

### 2.11.9 com_Sync()

| | | |
|---|---|---|
| **Syntax** | **com_Sync();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **status** | |
| | **status** | Returns **1** if the qualifier character has been received, else returns **0.** |
| | | |
| **Description** | If a ***qualifier*** character is specified when using buffered communications, after the buffer is initialised with **com_Init(..) ,** the service will ignore all characters until the ***qualifier*** is received and only then initiate the buffer write sequence with incoming data. com_Sync() is called to determine if the qualifier character has been received yet. | |
| | | |
| **Example** | `com_Sync(); // reset to polled mode` | |

### 2.11.10 com_TX(buf, bufsize)

| | | |
|---|---|---|
| **Syntax** | **com_TX(buf, bufsize);** | |
| | | |
| **Arguments** | **buf, bufsize** | |
| | **buf** | Specifies the address of a buffer automatically sent to the serial output in the background. |
| | **bufsize** | Specifies the byte count to be sent to the serial output. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **None** | |
| | | |
| **Description** | Sends a block of bytes from a user buffer to the serial port automatically. The progress of this process can be monitored with the com_TXcount function. Completion must be checked for using com_TXcount before the com_TX function can be re-issued, else the transmission will be corrupted. Also, the data in the buffer should not be changed while a transmission is in progress.. | |
| | | |
| **Example** | `com_TX(mybuf, 33); // send 33 bytes that are stored at mybuf` | |

### 2.11.11 com_TXcount()

| | | |
|---|---|---|
| **Syntax** | **com_TXcount();** | |
| | | |
| **Arguments** | **None** | |
| | | |
| **Returns** | **Count** | |
| | **Count** | Returns count of characters. |
| | | |
| **Description** | Return count of characters remaining that are being sent with the com_TX function . | |
| | | |
| **Example** | `com_TX(mybuf, 33); // send 33 bytes that are stored at mybuf`<br>`while(com1_TXCount()); //wait for the process to completely`<br>`com_TX(anotherbuf, 10); // send 10 bytes from another buffer` | |

## 2.11.12 com_CSUM_8(buf,count)

| | | |
|---|---|---|
| **Syntax** | **com_CSUM_8(buf,count);** | |
| | | |
| **Arguments** | **buf, count** | |
| | **buf** | Specifies the address of a buffer to be tallied for the checksum. |
| | **count** | Specifies the byte count to be tallied. |
| | The arguments can be a variable, array element, expression or constant. | |
| | | |
| **Returns** | **checksum** | |
| | **checksum** | the negated sum of **count** bytes in the buffer. |
| | | |
| **Description** | Given a pointer to a buffer and a byte count, calculate the 8bit LRC. if you calculate all of the incoming data INCLUDING a sent checksum, the result should be 0x00. This is equivalent to simple addition of all bytes and returning the negated sum an 8 bit value.<br>For the standard test string "123456789", **com_CSUM_8** will return 0x0023. | |
| | | |
| **Example** | `com_CSUM_8(mybuf, 33); // return the checksum of 33 bytes at mybuf` | |

### 2.11.13 com_TXcount()

| | | |
|---|---|---|
| **Syntax** | **com_TXcount();** | |
| | | |
| **Arguments** | **None** | |
| | | |
| **Returns** | **Count** | |
| | **Count** | Returns count of characters. |
| | | |
| **Description** | Return count of characters remaining that are being sent with the com_TX function | |
| | | |
| **Example** | `com_TX(mybuf, 33); // send 33 bytes that are stored at mybuf`<br>`while(com1_TXCount()); //wait for the process to completely`<br>`com_TX(anotherbuf, 10); // send 10 bytes from another buffer` | |

### 2.11.14 com_CRC_16(buf, count)

| | | |
|---|---|---|
| **Syntax** | **com_CRC_16(buf,count);** | |
| | | |
| **Arguments** | **buf, count** | |
| | **buf** | Specifies the address of a buffer to be tallied for the checksum. |
| | **count** | Specifies the byte count to be tallied. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **checksum** | |
| | **checksum** | The CRC16 using the polynomial x16 + x15 + x2 + 1, seed 0x0000 |
| | | |
| **Description** | Given a pointer to a buffer and a byte count, calculate CRC16. if you calculate all of the incoming data INCLUDING a sent checksum, the result should be 0x0000. This is equivalent to x16 + x15 + x2 + 1 with a seed of 0x0000.<br>For the standard test string "123456789", **com_CRC_16** will return 0xBB3D. | |
| | | |
| **Example** | `to(mybuf); putstr(“123456789”);`<br>`com_CRC_16(mybuf, 9); // return the CRC 0xBB3D` | |

### 2.11.15 com_CRC_MODBUS(buf, count)

| Syntax | com_CRC_MODBUS(buf,count); | |
|---|---|---|
| | | |
| **Arguments** | **buf, count** | |
| | **buf** | Specifies the address of a buffer to be tallied for the checksum. |
| | **count** | Specifies the byte count to be tallied. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **checksum** | |
| | **checksum** | The CRC16 using the polynomial x16 + x15 + x2 + 1, seed 0xFFFF |
| | | |
| **Description** | Given a pointer to a buffer and a byte count, calculate CRC16 for MODBUS. if you calculate all of the incoming data INCLUDING a sent checksum, the result should be 0x0000. This is equivalent to x16 + x15 + x2 + 1 with a seed of 0xFFFF.<br>For the standard test string "123456789", **com_CRC_MODBUS** will return 0x4B37. | |
| | | |
| **Example** | `to(mybuf); putstr("123456789");`<br>`com_CRC_MODBUS(mybuf, 9); // return the CRC 0x4B37` | |

### 2.11.16 com_CRC_CCITT(buf, count, seed)

| | | |
|---|---|---|
| **Syntax** | **com_CRC_CCITT(buf, count, seed)** | |
| | | |
| **Arguments** | **buf, count, seed** | |
| | **buf** | Specifies the address of a buffer to be tallied for the checksum. |
| | **count** | Specifies the byte count to be tallied. |
| | **seed** | CRC calculation starting value. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **checksum** | |
| | **checksum** | The CRC using the polynomial x16 + x12 + x5 + 1 with a given seed |
| | | |
| **Description** | Given a pointer to a buffer and a byte count, and a seed, calculate CCITT-CRC16.<br>If you calculate all of the incoming data INCLUDING a sent checksum, the result should be 0x0000. This is equivalent to x16 + x12 + x5 + 1 with a seed .<br>seed = 0 (XMODEM protocol) result = 0x31C3<br>seed = 0xFFFF, result = 0x29B1<br>seed = 0x1D0F, result = 0xE5CC | |
| | | |
| **Example** | `to(mybuf); putstr("123456789");`<br>`com_CRC_MODBUS(mybuf, 9, 0); // return the CRC 0x31C3` | |

### 2.11.17 sys_EventsPostpone()

| | |
|---|---|
| **Syntax** | **sys_EventsPostpone();** |
| | |
| **Arguments** | **none** |
| | |
| **Returns** | **nothing** |
| | |
| **Description** | Postpone the comms events for timeout or buffer full.<br>Postpones comms event until the sys_EventResume function is executed. The comms timer3 or comms buffer full event are postponed and will not take place until a **sys_EventResume** function is encountered. This function is required to allow a sequence of instructions or functions to occur that would otherwise be corrupted by an event occurring during the sequence of instructions or functions. A good example of this is when you set a position to print, if there was no way of locking the current sequence, an event may occur which does a similar thing, and a contention would occur - printing to the wrong position. This function should be used wisely, if any action that is required would take considerable time, it is better to gather what is needed quickly in the service function, and set a flag to tell the main program that further action is now required. |
| | |
| **Example** | `sys_EventsPostpone();  // stop any comms event for a short while`<br>`gfx_MoveTo(10,100); // print the current time of day`<br>`putstr(timestring);`<br>`sys_EvensResume();  //ok, resume events` |

### 2.11.18 sys_EventsResume()

| | |
|---|---|
| **Syntax** | **sys_EventsResume();** |
| | |
| **Arguments** | **none** |
| | |
| **Returns** | **nothing** |
| | |
| **Description** | Resume any pending comms timer3 or comms buffer full event. |
| | |
| **Example** | `sys_EventsPostpone();  // stop any comms event for a short while`<br>`gfx_MoveTo(10,100); // print the current time of day`<br>`putstr(timestring);`<br>`sys_EventsResume();   // ok, resume events` |

## 2.12 Sound and Tune (RTTTL) Functions

**Summary of Functions in this section:**

- beep(note, duration)
- tune_Play(tuneptr)
- tune_Pause()
- tune_Continue()
- tune_Stop()
- tune_End()
- tune_Playing()

### 2.12.1 beep(note, duration)

| | | |
|---|---|---|
| **Syntax** | **beep(note, duration);** | |
| | | |
| **Arguments** | **note, duration** | |
| | **note** | A value (usually a constant) specifying the frequency of the note. |
| | **duration** | specifies the time in milliseconds that the note will be played for. |
| | The arguments can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Simple utility to produce a single musical note for the required duration. | |
| | | |
| **Example** | `Beep(20, 50);    // play note 20 for 50 milliseconds` | |

### 2.12.2 tune_Play(tuneptr)

| Syntax | tune_Play(tuneptr); | |
|---|---|---|
| | | |
| **Arguments** | **tuneptr** | |
| | **tuneptr** | Specifies a pointer to a data statement or a string constant containing RTTTL information.<br>**Note**: The argument passed to the tune_Play(...) function must be an ASCII string. If the string is passed as a pointer from a #DATA statement, it must be terminated with a zero (0x00). if a string is passed directly as a parameter, the '0' is automatically appended by the compiler as per normal strings. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | The tune_Play(...) function in 4DGL uses a variant of the "Ring Tone Text Transfer Language" (RTTTL) developed by Nokia for cellphone ring tones. There are certain differences that need to be taken into account, and several additions that will be described later. It is suggested that you have a look at the original format first, one suggestion being the excellent description on the web at:<br>http://www.activexperts.com/xmstoolkit/sms/rtttl/<br>and<br>http://en.wikipedia.org/wiki/Ring_Tone_Transfer_Language<br>You will find that with a little practice and minor modifications, most RTTTL tunes that can be downloaded off the web are playable with the **tune_Play(...)** function. Also, a wide range of sound effects can be made using standard RTTTL notation augmented with the additional 4DGL functions.<br>**The 4DGL implementation:**<br>• The "**b=nnn**" in 4DGL does not represent "beats per minute" (bpm), it represents "milliseconds per hemidemisemiquaver".<br>e.g. 120 bpm is 2 beats per second = 128 demisemiquavers per second which is 7.8125msec per hemidemisemiquaver. Conversely, the default 4DGL value for b = 16msec per hemidemisemiquaver equates to 62.5 bpm.<br>• The argument passed to the tune_Play(...); command must be a string. If the string is passed as a pointer from a **#DATA** statement, it must be terminated with a zero (0x00).<br>(if a string is passed directly as a parameter, the zero (0x00) is automatically appended by the compiler as per normal strings).<br>• The original RTTTL format is a string divided into three sections:<br>**name, default value, data.**<br>The 4DGL implementation does not have the "name" section - this would be just a waste of space.<br>• The 4DGL implementation does not require any spaces or colons anywhere, once | |

again this would be a waste of space.

- The 4DGL implementation allows default values to be changed anywhere in the string and does not need to be at the start.
- The optional default modifiers is a set of parameters separated by commas, where each value contains a key and a value separated by an '=' character, which describes certain defaults which will be adhered to during the execution of the ringtone string.
    - **d - duration**
      The default duration can be one of 1, 2, 4, 8, 16, 32 or 64 (64 = 1/64th, 1 = 1 whole unit)
      **1** specifies a Semibreve (Whole Note),
      **2** indicates it a Minim (Half Note),
      **4** is a Crotchet (Quarter Note) etc up to **64** which is a hemidemisemiquaver (64th note).
    - **b - beat/tempo**
      "milliseconds per demisemiquaver"
    - **o - octave**
      The default octave (scale) can be 4, 5, 6, or 7.
    - **If not specified, defaults are:**
      **duration = 4** (same as d=4)
      **octave = 6** (same as o=6)
      **beat = 16** (same as b=16) close to 63bpm

**4DGL extended default values:**

- **r -** set repeat point and counter (eg r=4)
  min = 2, max = 255
  default value = forever
- **p -** set portamento value (eg p=5)
  min = 1, max = 14
  default value is 4
- **a -** set arpeggiation step value (eg a=1)
  min = 1, max = 16
  default value is 1

**4DGL extended commands associated with extended default values:**

- **R** execute a repeat specified by r =
  Note: if no repeat count has been specified, the string will repeat forever
- **{** turn portamento ON
- **}** turn portamento OFF
  Note: portamento default value is OFF
- **+** raise note as specified by arpeggiation step value
- **-** lower note as specified by arpeggiation step value

**Example**

```
/*
This example shows how to use the RTTTL tunes to
```

```
generate complex sounds and music.
*/

//-------------------------------------------------------------------
#DATA
   // b=250
   byte Muppets   "d=4,o=5,b=15,",
                  "c6,c6,a,b,8a,b,g,p,c6,c6,a,8b,8a,8p,g.,p,e,e,g,f,
                   8e,f,8c6,8c,8d,e,8e,8e,8p,8e,g,2p,c6,",
                  "c6,a,b,8a,b,g,p,c6,c6,a,8b,a,g.,p,e,e,g,f,8e,f,
                   8c6,8c,8d,e,8e,d,8d,c",0

   // part of haunted house theme
   byte HauntedHouse  "d=4,o=5,b=20,",
                      "2a4,2e,2d#,2b4,2a4,2c,2d,2a#4,2e.,e,1f4,1a4,
                       1d#,2e.,d,2c.,b4,1a4", 0

   // simple scale with default settings
   byte SimpleScale   "c,d,e,f,g,a,b,c7", 0

   // simple scale with default settings and portamento use.
   // Note the portamento speed change in the middle of the string,
   // and the curly braces that turn the portamento on and off.
   byte SimpleScaleP   "b=50,{,c,d,e,f,p=7,g,a,},b,c7", 0

   // simple scale, much faster
   // note b=20 as default, so each note plays for 20msec when d=64
   byte Scale2        "d=64,c,d,e,f,g,a,b,c7", 0

   // simple scale, much faster - with a repeat command set to 20
   // note b=20 as default, so each note plays for 20msec when d=64,
   // and we repeat 20 times
   byte ScaleRep      "d=64,r=20,c,d,e,f,g,a,b,c7,R", 0

   // simple scale, at the fastest possible rate, repeat 200 times
   // note that b=1 and d=64 so each note plays for only 1msec
   byte ScaleRep2     "b=1,d=64,r=200,c,d,e,f,g,a,b,c7,R", 0

   // simple scale using appregiation to increment the note step
   // note that commas can be left out to save space if there is no
   // indecision about delimit value
   byte ApprScale   "a=1,c,+++++++++++------------", 0

   // scale using appregiation to increment the note step, and the
   // note step is larger
   // note that commas can be left out to save space if there is no
   // indecision about delimit value
   byte ApprScaleF   "d=8,a=4,c,++++++++++++------------", 0

   // same as above but demonstrates repeating instead of multiple
   // inc/dec operators
   // note that commas can be left out to save space if there is no
   // indecision about delimit value
   byte ApprScaleFR   "d=8,a=4,c5,r=11,+,R,r=11,-,R", 0

   // you can build your own scale sequencers
   byte COMPLEX_C         "d=64,a=5,c4,r=8,+,R", 0
   byte COMPLEX_DSHARP    "d=64,a=5,d#4,r=8,+,R", 0
```

|  |  |
|---|---|
|  | (code below) |

```
   byte COMPLEX_G         "d=64,a=5,g4,r=8,+,R", 0

   // just having a bit of fun
   byte DEMO     "a=3,p=3,o=5,d=4,b=5,
                 {,a,r=20,+,R,},c,d=16,a=5,r=50,-,R, R",0  // forever
#END
//-------------------------------------------------------------------

#constant number_of_examples 13
var examples[number_of_examples];
var names[number_of_examples];

//-------------------------------------------------------------------
func main()
   var n;


   // lookup table for the examples
   examples[0] := HauntedHouse;
   examples[1] := SimpleScale;
   examples[2] := SimpleScaleP;
   examples[3] := Scale2;
   examples[4] := ScaleRep;
   examples[5] := ScaleRep2;
   examples[6] := ApprScale;
   examples[7] := ApprScaleF;
   examples[8] := ApprScaleFR;
   examples[9] := COMPLEX_C;
   examples[10] := COMPLEX_DSHARP;
   examples[11] := COMPLEX_G;
   examples[12] := Muppets;

   // lookup table for the example names
   names[0] := "HauntedHouse";
   names[1] := "SimpleScale";
   names[2] := "SimpleScaleP";
   names[3] := "Scale2";
   names[4] := "ScaleRep";
   names[5] := "ScaleRep2";
   names[6] := "ApprScale";
   names[7] := "ApprScaleF";
   names[8] := "ApprScaleFR";
   names[9] := "COMPLEX_C";
   names[10] := "COMPLEX_DSHARP";
   names[11] := "COMPLEX_G";
   names[12] := "Muppets";

   repeat
      n := 0;
      // play each demo, demonstrate multitasking while tune playing
      repeat
         gfx_Cls();
         txt_MoveCursor(0,8);
         tune_Play( examples[n] );
         txt_Set(TEXT_PRINTDELAY, 0);
         putstr( names[n++] );

         repeat
```

```
            txt_Set(TEXT_PRINTDELAY, 50);
            txt_MoveCursor(0,0);
            putstr("Playing");
            pause(150);
            txt_MoveCursor(0,0);
            putstr("       ");
         until (!(sys_Get(CONTROL) & PLAYING));// wait until the tune
                                               // string finishes.
         pause(1000);   // then pause 5 seconds
      until (n == number_of_examples);

   gfx_Cls();
   txt_Set(TEXT_PRINTDELAY, 0);
   tune_Play( DEMO );                  // last example plays forever
   putstr( "DEMO CONTINUOUS" );

   // the last demo endlessly loops, play for 10 seconds then pause
   pause(10000);

   tune_Pause();
   print("\nPaused....");

   pause(10000);             // pause for 10 seconds

   tune_Continue();          // continue
   print("\nContinue....");

   pause(10000);             // for 10 seconds

   tune_End();               // then end it
   print("\nEnd....");

   pause(10000);             // wait for 10 seconds

   forever                   // then do it all again

endfunc
//---------------------------------------------------------------------
```

### 2.12.3 tune_Pause()

| | |
|---|---|
| **Syntax** | **tune_Pause();** |
| | |
| **Arguments** | **none** |
| | |
| **Returns** | **nothing** |
| | |
| **Description** | Suspends any current tune from playing until a **tune_Continue()**, **tune_Stop()** or a new **tune_Play("...")** function is called. The oscillator is not stopped. |
| | |
| **Example** | `See example in tune_Play(..)` |

### 2.12.4 tune_Continue()

| | |
|---|---|
| **Syntax** | **tune_Continue();** |
| | |
| **Arguments** | **none** |
| | |
| **Returns** | **nothing** |
| | |
| **Description** | Continues playing any previously stopped or paused tune. |
| | |
| **Example** | `See example in tune_Play(..)` |

### 2.12.5 tune_Stop()

| | |
|---|---|
| **Syntax** | **tune_Stop();** |
| | |
| **Arguments** | **none** |
| | |
| **Returns** | **nothing** |
| | |
| **Description** | Pauses a tune and silences the oscillator until a **tune_Continue()**, **tune_Stop()**, **tune_End()** or a new **tune_Play("...")** function is called. |
| | |
| **Example** | `See example in tune_Play(..)` |

## 2.12.6 tune_End()

| | |
|---|---|
| **Syntax** | **tune_End();** |
| | |
| **Arguments** | **none** |
| | |
| **Returns** | **nothing** |
| | |
| **Description** | Ends any current tune and resets the tune interpreter. |
| | |
| **Example** | `See example in tune_Play(..)` |

### 2.12.7 tune_Playing()

| | | |
|---|---|---|
| **Syntax** | **tune_Playing();** | |
| | | |
| **Arguments** | **none** | |
| | | |
| **Returns** | **state** | |
| | **state** | Returns: **1** if a tune is playing<br>Returns: **0** if no tune is playing |
| | | |
| **Description** | Use this function to check for any current tunes being played. Returns 1 if tune is playing, 0 if no tune is playing. | |
| | | |
| **Example** | `See example in tune_Play(..)` | |

## 2.13 General Purpose Functions

**Summary of Functions in this section:**

- pause(time)
- lookup8 (key, byteConstList )
- lookup16 (key, wordConstList )

### 2.13.1 pause(time)

| | | |
|---|---|---|
| **Syntax** | **pause(time);** | |
| | | |
| **Arguments** | **time** | |
| | **time** | A value specifying the delay time in milliseconds. |
| | The argument can be a variable, array element, expression or constant | |
| | | |
| **Returns** | **nothing** | |
| | | |
| **Description** | Stop execution of the user program for a predetermined amount of time. | |
| | | |
| **Example** | `if (joystick() == FIRE)      // if fire button pressed`<br>`    pause(30)                 // slow down the loop`<br>`else`<br>`    ...` | |

### 2.13.2 lookup8(key, byteConstList)

| **Syntax** | **lookup8(key, byteConstList);** | |
|---|---|---|
| | | |
| **Arguments** | **key, byteConstList** | |
| | **key** | A byte value to search for in a fixed list of constants. The **key** argument can be a variable, array element, expression or constant |
| | **byteConstList** | A comma separated list of constants and strings to be matched against **key.**<br>**Note:** the string of constants may be freely formed, see example. |
| | | |
| **Returns** | **result** | |
| | result | See description. |
| | | |
| **Description** | Search a list of 8 bit constant values for a match with a search value **key**. If found, the index of the matching constant is returned in **result,** else **result** is set to zero. Thus, if the value is found first in the list, **result** is set to one. If second in the list, **result** is set to two etc. If not found, **result** is returned with zero.<br>**Note:** The list of constants cannot be re-directed. The lookup8(...) functions offer a versatile way for returning an index for a given value. This can be very useful for data entry filtering and parameter input checking and where ever you need to check the validity of certain inputs. The entire search list field can be replaced with a single name if you use the $ operator in constant, eg :<br>`#constant HEXVALUES $"0123456789ABCDEF"` | |
| | | |

**Example**

```
func main()
    var key, r;

    key := 'a';
    r := lookup8(key, 0x4D, "abcd", 2, 'Z', 5);
    print("\nSearch value 'a' \nfound as index ", r)

    key := 5;
    r := lookup8(key, 0x4D, "abcd", 2, 'Z', 5);
    print("\nSearch value 5 \nfound at index ", r)
    putstr("\nScanning..\n");

    key := -12000; // we will count from -12000 to +12000, only
                   // the hex ascii values will give a match value

    while(key <= 12000)
        r := lookup8(key, "0123456789ABCDEF" ); // hex lookup
        if(r) print([HEX1] r-1); // only print if we got a match in
                                 // the table
        key++;
    wend

    repeat forever
endfunc
```

### 2.13.3 lookup16(key, wordConstList)

| | | |
|---|---|---|
| **Syntax** | **lookup16(key, wordConstList);** | |
| | | |
| **Arguments** | **key, wordConstList** | |
| | **key** | A word value to search for in a fixed list of constants. The **key** argument can be a variable, array element, expression or constant |
| | **wordConstList** | A comma separated list of constants to be matched against **key.** |
| | | |
| **Returns** | **result** | |
| | result | See description. |
| | | |
| **Description** | Search a list of 16 bit constant values for a match with a search value **key**. If found, the index of the matching constant is returned in **result,** else **result** is set to zero. Thus, if the value is found first in the list, **result** is set to one. If second in the list, **result** is set to two etc. If not found, **result** is returned with zero.<br>**Note:** The lookup16(...) functions offer a versatile way for returning an index for a given value. This is very useful for parameter input checking and where ever you need to check the validity of certain values. The entire search list field can be replaced with a single name by using the $ operator in constant, eg:<br>`#constant LEGALVALS  $5,10,20,50,100,200,500,1000,2000,5000,10000` | |
| | | |
| **Example** | | |

```
func main()
    var key, r;

    key := 5000;
    r := lookup16(key, 5,10,20,50,100,200,500,1000,2000,5000,10000);
    //r := lookup16(key, LEGALVALS);

    if(r)
        print("\nSearch value 5000 \nfound at index ", r);
    else
        putstr("\nValue not found");
    endif

    print("\nOk");  // all done

    repeat forever
endfunc
```

## 3. GOLDELOX-PoGa EVE System Registers Memory Map

The following tables outline in detail the GOLDELOX-PoGa system registers.

### Table 3.1: BYTE-Size Registers Memory Map. Accessible with PeekB and PokeB

| LABEL | ADDRESS | | USAGE |
|---|---|---|---|
| | DEC | HEX | |
| CLIP_LEFT_POS | 128 | 0x80 | left clipping point (set with gfx_ClipWindow(...) |
| CLIP_TOP_POS | 129 | 0x81 | top clipping point (set with gfx_ClipWindow(...) |
| CLIP_RIGHT_POS | 130 | 0x82 | right clipping point (set with gfx_ClipWindow(...) |
| CLIP_BOTTOM_POS | 131 | 0x83 | bottom clipping point (set with gfx_ClipWindow(...) |
| CLIP_LEFT | 132 | 0x84 | left clip point active(reads full size if clipping turned off) |
| CLIP_TOP | 133 | 0x85 | top clip point active(reads full size if clipping turned off) |
| CLIP_RIGHT | 134 | 0x86 | right clip point active(reads full size if clipping turned off) |
| CLIP_BOTTOM | 135 | 0x87 | bottom clip point active(reads full size if clipping turned off) |
| FONT_TYPE | 136 | 0x88 | 0 = system font, else pointer to user font |
| FONT_MAX | 137 | 0x89 | number of chars in font set |
| FONT_OFFSET | 138 | 0x8A | ASCII offset (usually 0x20) |
| FONT_WIDTH | 139 | 0x8B | width of font (pixel units) |
| FONT_HEIGHT | 140 | 0x8C | height of font (pixel units) |
| TEXT_XMAG | 141 | 0x8D | text width magnification |
| TEXT_YMAG | 142 | 0x8E | text height magnification |
| TEXT_MARGIN | 143 | 0x8F | left column for carriage return |
| TEXT_DELAY | 144 | 0x90 | print delay (0-255msec) |
| TEXT_X_GAP | 145 | 0x91 | X pixel gap between chars |
| TEXT_Y_GAP | 146 | 0x92 | Y pixel gap between chars |
| GFX_XMAX | 147 | 0x93 | current display width-1 (always 127 for PoGa) |
| GFX_YMAX | 148 | 0x94 | current display height-1 (always 127 for PoGa) |
| GFX_SCREENMODE | 149 | 0x95 | Current screen mode (0-3) |
| GFX_STRINGWIDTH | 150 | 0x96 | Width (in pixels) after last string width calculation. |
| GFX_STRINGHEIGHT | 151 | 0x97 | Height (in pixels) after last string height calculation. |
| IMAGE_DELAY | 152 | 0x98 | 0 if image, frame delay (if animation) |
| IMAGE_MODE | 153 | 0x99 | **Bit 4**: colour mode, other bits reserved (always '1', on PoGa 16bit colour only) |
| reserved | 154-159 | 0x9A-0x9F | reserved |

| Table 3.2: WORD-Size Registers Memory Map. Accessible with PeekW and PokeW | | | | |
|---|---|---|---|---|
| LABEL | ADDRESS | | USAGE | NOTES |
| | DEC | HEX | | |
| VM_OVERFLOW | 80 | 0x50 | 16bit overflow of 32bit results (see OVF() funtion) | SYSTEM |
| VM_COLOUR | 81 | 0x51 | internal variable for colour | SYSTEM |
| VM_RETVAL | 82 | 0x52 | return value of last function | SYSTEM |
| GFX_BACK_COLOUR | 83 | 0x53 | screen background colour | SYSTEM |
| GFX_OBJECT_COLOUR | 84 | 0x54 | graphics object colour | SYSTEM |
| GFX_TEXT_COLOUR | 85 | 0x55 | text foreground colour | SYSTEM |
| GFX_TEXT_BGCOLOUR | 86 | 0x56 | text background colour | SYSTEM |
| GFX_OUTLINE_COLOUR | 87 | 0x57 | circle/rectangle outline | SYSTEM |
| GFX_TRANSPARENT_COLOUR | 88 | 0x58 | Screen background colour | SYSTEM |
| GFX_LINE_PATTERN | 89 | 0x59 | line draw tessellation | SYSTEM |
| MEDIA_HEAD | 90 | 0x5A | media sector head position | SYSTEM |
| SYS_OSTREAM | 91 | 0x5B | Output stream handle | SYSTEM |
| GFX_LEFT | 92 | 0x5C | virtual left point for current image | SYSTEM |
| GFX_TOP | 93 | 0x5D | virtual top point for current image | SYSTEM |
| GFX_RIGHT | 94 | 0x5E | virtual right point for current image | SYSTEM |
| GFX_BOTTOM | 95 | 0x5F | virtual bottom point for current image | SYSTEM |
| GFX_X1 | 96 | 0x60 | image left clipped point | SYSTEM |
| GFX_Y1 | 97 | 0x61 | image top clipped point | SYSTEM |
| GFX_X2 | 98 | 0x62 | image right clipped point | SYSTEM |
| GFX_Y2 | 99 | 0x63 | image bottom clipped point | SYSTEM |
| GFX_X_ORG | 100 | 0x64 | current X origin | SYSTEM |
| GFX_Y_ORG | 101 | 0x65 | current Y origin | SYSTEM |
| RANDOM_LO | 102 | 0x66 | random number generator LO word | SYSTEM |
| RANDOM_HI | 103 | 0x67 | random number generator HI word | SYSTEM |
| MEDIA_ADDR_LO | 104 | 0x68 | media absolute byte address LO | SYSTEM |
| MEDIA_ADDR_HI | 105 | 0x69 | media absolute byte address HI | SYSTEM |
| SECTOR_ADDR_LO | 106 | 0x6A | media sector address LO | SYSTEM |
| SECTOR_ADDR_HI | 107 | 0x6B | media sector address HI | SYSTEM |
| SYSTEM_TIMER_LO | 108 | 0x6C | 1msec 32 bit free running timer LO word | SYSTEM |
| SYSTEM_TIMER_HI | 109 | 0x6D | 1msec 32 bit free running timer HI word | SYSTEM |

| | | | | |
|---|---|---|---|---|
| TIMER0 | 110 | 0x6E | 1msec user timer 0 | SYSTEM |
| TIMER1 | 111 | 0x6F | 1msec user timer 1 | SYSTEM |
| TIMER2 | 112 | 0x70 | 1msec user timer 2 | SYSTEM |
| TIMER3 | 113 | 0x71 | 1msec user timer 3 | SYSTEM |
| TIMER4 | 114 | 0x72 | User timer4 (shared with media timout) | SYSTEM |
| IMG_WIDTH | 115 | 0x73 | width of currently loaded image | SYSTEM |
| IMG_HEIGHT | 116 | 0x74 | height of currently loaded image | SYSTEM |
| IMG_FRAME_COUNT | 117 | 0x75 | count of frames in animation | SYSTEM |
| IMG_PIXEL_COUNT | 118 | 0x76 | pixel count of current object (LO word) | SYSTEM |
| IMG_PIXEL_COUNT_HI | 119 | 0x77 | pixel count of current object (HI word) | SYSTEM |
| USR_SP | 128 | 0x80 | EVE user defined stack pointer | STACK |
| USRVARS | 129 | 0x81 | EVE user variables VARS[255] | STACK |
| USRSTACK | 384 | 0x180 | EVE machine stack STACK[128] | STACK |

| **NOTES:** | |
|---|---|
| **SYSTEM** | SYSTEM registers are maintained by internal system functions and should not be written to. They should only ever be read.<br>**DO NOT WRITE to these registers.** |
| **STACK** | 128 word EVE system stack (STACK grows upwards) |

## 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.

4D Labs 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 and its suppliers specifically disclaim any expressed or implied warranty of fitness for High Risk Activities.

Use of 4D Labs’ 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 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..