4D LABS

GOLDELOX-GFX2
Internal 4DGL Functions

Document Date: 19th March 2010
Document Revision: 2.0

Table of Contents

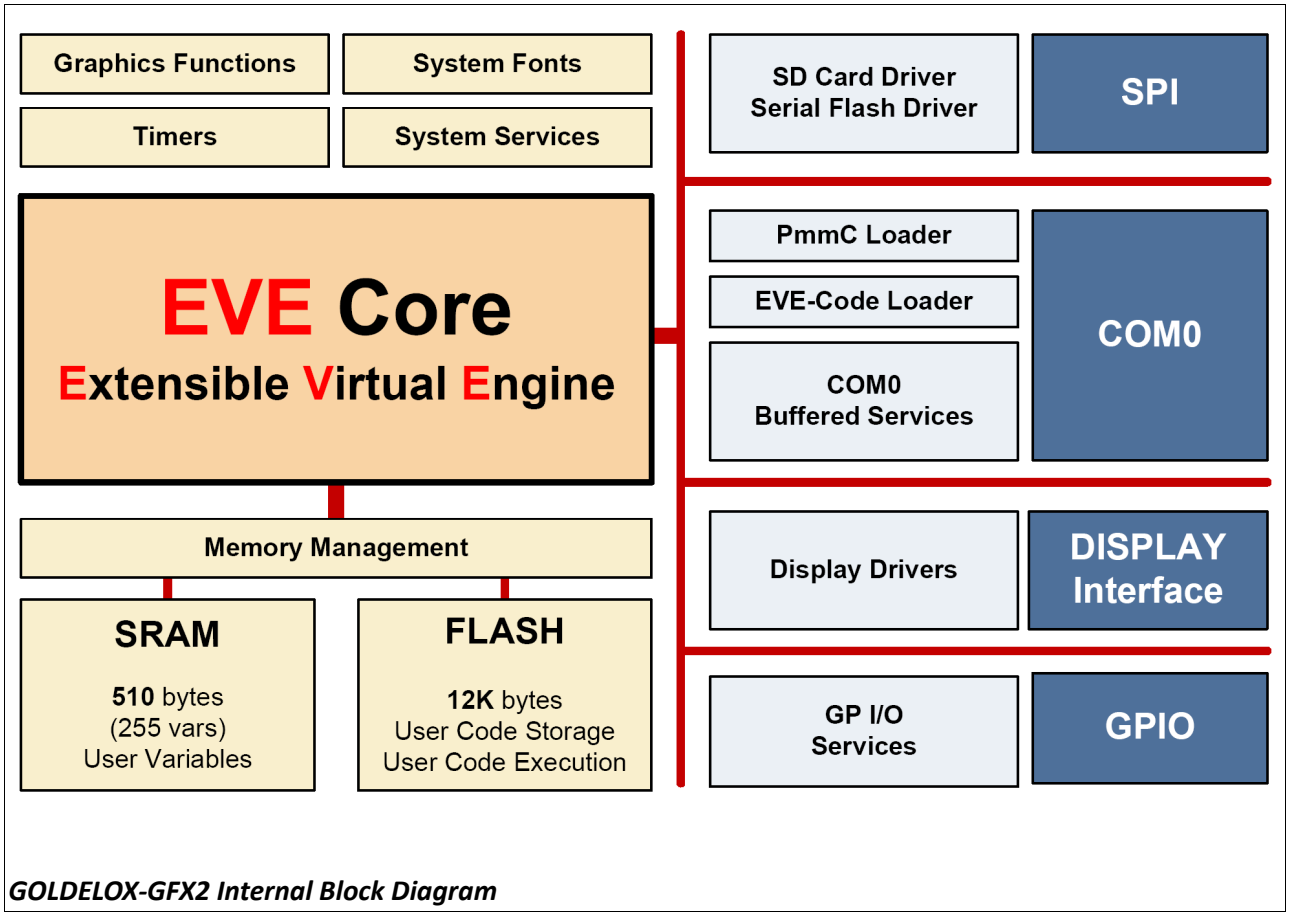
1. 4DGL Introduction

The 4D-Labs family of embedded graphics processors such as the : 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-GFX2. This document should be used in conjunction with “4DGL-Programmers-Reference-Manual” document.



GOLDELOX-GFX2 Internal Block Diagram

2. GOLDELOX-GFX2 Chip-Resident Functions Summary

The following is a summary of chip-resident 4DGL functions within the GOLDELOX-GFX2 graphics controller. The document is made up of the following sections:

2.1 GPIO Functions:

- pin_Set(mode, pin)
  - OUTPUT, INPUT, ANALOGUE_8, ANALOGUE_10, ONEWIRE, SOUND
- pin_HI(pin)
- pin_LO(pin)
- pin_Read(pin)
- joystick()
- OW_Reset()
- OW_Read()
- OW_Read9()
- OW_Write(data)

2.2 Memory Access Functions:

- peekB(address)
- peekW(address)
- pokeB(address, byte_value)
- pokeW(address, word_value)
- bits_Set(address, mask)
- bits_Clear(address, mask)
- bits_Flip(address, mask)
- bits_Test(address, mask)

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_IncX()
- gfx_IncY()
- gfx_LineTo(xpos, ypos)
- gfx_LineRel(xpos, ypos)
- gfx_BoxTo(x2, y2)
- gfx_SetClipRegion()
- gfx_ClipWindow(x1, y1, x2, y2)
- gfx_FocusWindow()
- gfx_Set(function, value)

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

2.7 Display I/O Functions:

- disp_Init(initTable, stateMachine)
- disp_WriteControl(value)
- disp_WriteByte(value)
- disp_WriteWord(value)
- disp_ReadByte()
- disp_ReadWord()
- disp_BlitPixelFill(colour, count)
- disp_BlitPixelsToMedia()
- disp_BlitPixelsFromMedia(pixelcount)
- disp_SkipPixelsFromMedia(pixelcount)
- disp_BlitPixelsToCOM()
- disp_BlitPixelsFromCOM(mode)

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)

2.9 Flash Memory chip Functions:

- flash_SIG()
- flash_ID()
- flash_BulkErase()
- flash_BlockErase(blockAddress)

2.10 SPI Control Functions:

- spi_Init(speed, input_mode, output_mode)
- spi_Read()
- spi_Write(byte)
- spi_Disable()

2.11 Serial (UART) Communications Functions:

- serin()
- serout(char)
- setbaud(rate)
- com_AutoBaud(timeout)
- com_Init(buffer, buffsize, qualifier)
- com_Reset()
- com_Count()
- com_Full()
- com_Error()
- com_Sync()
- com_Checksum()
- com_PacketSize()

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 GPIO Functions

Summary of Functions in this section:

- pin_Set(mode, pin)
  - OUTPUT, INPUT, ANALOGUE_8, ANALOGUE_10, ONEWIRE, SOUND
- pin_HI(pin)
- pin_LO(pin)
- pin_Read(pin)
- joystick()
- OW_Reset()
- OW_Read()
- OW_Read9()
- OW_Write(data)

2.1.1 pin_Set(mode, pin)

Syntax	pin_Set(mode, pin);	
Arguments	**mode, pin**	
	mode	A value (usually a constant) specifying the pin operation.
	pin	A value (usually a constant) specifying the pin number.
	The arguments can be a variable, array element, expression or constant.	
Returns	**nothing**	
Description	GOLDELOX-GFX2 has limited but powerful I/O. There are pre-defined constants for **mode** and **pin**:	

pin constants	**pin** value
IO1	0
IO2	1

mode constants	**mode** value	meaning	IO1	IO2
OUTPUT	0	Pin is set to an output	YES	YES
INPUT	1	Pin is set to an input	YES	YES
ANALOGUE_8	2	Pin is set to analogue input, 8 bit mode	YES	NO
ANALOGUE_10	3	Pin is set to analogue input, 10 bit mode	YES	NO
ONEWIRE	4	Pin is set as Dallas One Wire I/O mode	YES	YES
SOUND	5	Pin is set for RTTTL sound output	YES	YES

Example

```
pin_Set(OUTPUT, IO2);         // set IO2 to be used as an output
pin_Set(ANALOGUE_10, IO1);    // set IO1 to be used as analogue input
```

2.1.2 pin_HI(pin)

Syntax	**pin_HI(pin);**	
Arguments	**pin**	
	pin	A value (usually a constant) specifying the pin number.
	The arguments can be a variable, array element, expression or constant.	
Returns	**nothing**	
Description	Outputs a "High" level (logic 1) on the appropriate pin that was previously selected as an Output. If the pin is not already set to an output, it is automatically made an output.	
Example	`pin_HI(IO2);  // output a Logic 1 on IO2 pin`	

2.1.3 pin_LO(pin)

Syntax	**pin_LO(pin);**	
Arguments	**pin**	
	pin	A value (usually a constant) specifying the pin number.
	The arguments can be a variable, array element, expression or constant.	
Returns	**nothing**	
Description	Outputs a "Low" level (logic 0) on the appropriate pin that was previously selected as an Output. If the pin is not already set to an output, it is automatically made an output.	
Example	`pin_LO(IO1);  // output a Logic 0 on IO1 pin`	

2.1.4 pin_Read(pin)

Syntax	**pin_Read(pin);**	
Arguments	**pin**	
	pin	A value (usually a constant) specifying the pin number.
	The arguments can be a variable, array element, expression or constant.	
Returns	**value**	
	value	Returns a Logic 1 (0x0001) or a Logic 0 (0x0000) or the analogue value of the input pin.
Description	Reads the logic state or the analogue value of the pin that was previously selected as an Input. Returns a "Low" (logic 0) or "High" (logic 1) or Analogue value n.	
Example	`if(pin_Read(IO1) < 200)  // read the analogue value on IO1` `  calc_Threshold();` `else` `  ...`	

2.1.5 joystick()

Syntax	**joystick();**	
Arguments	**none**	
Returns	**value**	
	value	Returns the joystick value.
Description	Returns the value of the Joystick position (5 position switch implementation). The JOYSTICK values are: **Value** 0 1 2 3 4 5 **Status** Released UP LEFT DOWN RIGHT FIRE **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. Refer to the GOLDELOX-GFX2 data sheet example schematics for the required resistor values.	
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("FIRE");`	

2.1.6 OW_Reset()

Syntax	OW_Reset();	
Arguments	**none**	
Returns	**result**	
	result	Reset, and returns the status of the ONEWIRE device **0** = ACK **1** = No Activity (refer to Dallas 1wired documentation for further information)
Description	Resets a ONEWIRE device and returns the status.	
Example	`print ("result=", OW_Reset());` This example will print a 0 if the device initialised successfully.	

2.1.7 OW_Read()

Syntax	OW_Read();	
Arguments	**none**	
Returns	**value**	
	value	A word holding the lower 8 bits contain data bits received from the 1-Wire device.
Description	Reads the 8 bit value from a 1-Wire devices register. (refer to Dallas 1wired documentation for further information)	
Example	`// read temperature from DS1821 device` `var temp_buf;` `OW_Reset();  // reset the device` `OW_Write(0xAA);  // send the read command` `temp_buf := OW_Read();  // read the device register`	

2.1.8 OW_Read9()

Syntax	OW_Read9();	
Arguments	none	
Returns	value	
	value	A word holding 9 or more data bits received from the 1-Wire device.
Description	Reads the 9 or more bit value from a 1-Wire devices register. (refer to Dallas 1wired documentation for further information)	
Example	`// read temperature from DS1821 device` `var temp_buf;` `OW_Reset();  // reset the device` `OW_Write(0xAA);  // send the read command` `temp_buf := OW_Read9();  // read the device register`	

2.1.9 OW_Write(data)

Syntax	**OW_Write(data);**	
Arguments	**data**	
	data	The lower 8 bits of ***data*** are sent to the 1-Wire device.
	The argument can be a variable, array element, expression or constant.	
Returns	**nothing**	
Description	Writes the 8 bit data to 1-Wire devices register. (refer to Dallas 1wired documentation for further information)	
Example		

```
//=====================================================================
// For this demo to work, a Dallas DS1821 must be connected to
// IO1 AND POWERED FROM 5V.
// DS1821 pin1 = Gnd / pin2 = data in/out / pin 3 = +5v
// Refer to the Dallas DS1821 for further information
//=====================================================================
var temp_buf, stat_buf;
func main()
   pause(1000);
   txt_MoveCursor(0,0);
   pin_Set(ONEWIRE, PIN_1);     // set either I/O pin to 1 wire mode
   if(OW_Reset())               // initialise and test
      print("No device detected");
      while(1);
   endif
   txt_Set(TEXT_COLOUR, LIGHTGREY);
   txt_Set(FONT_SIZE, FONT_LARGE);
   // refer to data sheet for continuous/polled mode
   // OW_Write(0x0C);                  // write status
   // OW_Write(0b01000010);            // set continuous conversion

   repeat
      txt_MoveCursor(0, 0);
      print ("result=", OW_Reset());
      OW_Write(0xEE);                 // start conversion
      OW_Reset();                     // reset
      OW_Write(0xAA);                 // get temperature
      temp_buf := OW_Read();
      OW_Reset();                     // optional
      OW_Write(0xAC);                 // optional read status
      stat_buf := OW_Read();          // optional 82 when DS1821 run

      txt_MoveCursor(1, 0);
      print ("temp_buf=0x", [HEX2] temp_buf);
      txt_MoveCursor(2, 0);
      print ("stat_buf=0x", [HEX2] stat_buf);
   forever
endfunc
```

2.2 Memory Access Functions

Summary of Functions in this section:

- peekB(address)
- peekW(address)
- pokeB(address, byte_value)
- pokeW(address, word_value)
- bits_Set(address, mask)
- bits_Clear(address, mask)
- bits_Flip(address, mask)
- bits_Test(address, mask)

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 arguments 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.** **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;` `myvar := peekB(GFX_XMAX) + 1;` 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 arguments 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;` `myvar := peekW(SYSTEM_TIMER_LO);` This example places the low word of the 32 bit system timer in **myvar**. The equivalent operation using a pointer is:- 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.** **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);` 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);` This example sets TIMER2 to 5 seconds. The equivalent operation using a pointer is: `*TIMER2 := 5000;`	

2.2.5 bits_Set(address, mask)

Syntax	**bits_Set(address, mask);**	
Arguments	**address, mask**	
	address	The address of a user memory location.
	mask	The 16 bit mask containing bits to be set.
	The arguments can be a variable, array element, expression or constant.	
Returns	**nothing**	
Description	This function sets the required bits at address by 'ORing' the mask with the value stored at a**ddress**. **Note:** the bits_Set, bits_Clear, bits_Flip and bits_Test functions can only be used for user memory and will not work with system register variables	
Example	`var myval;` `myval := 3;` `bits_Set(myval, 0xC0);` `print([HEX], myval);` This example sets bits 6 and 7 of myval	

2.2.6 bits_Clear(address, mask)

Syntax	**bits_Clear(address, mask);**	
Arguments	**address, mask**	
	address	The address of a user memory location.
	mask	The 16 bit mask containing bits to be cleared.
	The arguments can be a variable, array element, expression or constant.	
Returns	**nothing**	
Description	This function clears the required bits at address by 'ANDing' the inverted mask with the value stored at a**ddress**. **Note:** the bits_Set, bits_Clear, bits_Flip and bits_Test functions can only be used for user memory and will not work with system register variables.	
Example	`var myval;` `myval := 0xFFFF;` `bits_Clear(myval, 0x3C00);` `print([HEX], myval);` This example clears bits 10, 11, 12 and 13 of myval	

2.2.7 bits_Flip(address, mask)

Syntax	**bits_Flip(address, mask);**	
Arguments	**address, mask**	
	address	The address of a user memory location.
	mask	The 16 bit mask containing bits to be flipped.
	The arguments can be a variable, array element, expression or constant.	
Returns	**nothing**	
Description	This function flips the required bits at address by 'XORing' the mask with the value stored at a**ddress**. **Note:** the bits_Set, bits_Clear, bits_Flip and bits_Test functions can only be used for user memory and will not work with system register variables.	
Example	`var myval;` `myval := 0xFFFF;` `bits_Flip(myval, 0x8802);` `print([HEX], myval);` This example clears bits 15, 11, and 1 of myval	

2.2.8 bits_Test(address, mask)

Syntax	**bits_Test(address, mask);**	
Arguments	**address, mask**	
	address	The address of a user memory location.
	mask	The 16 bit mask containing bits to be tested.
	The arguments can be a variable, array element, expression or constant.	
Returns	**result**	
	result	Returns: - TRUE (logic 1) if any of the tested bits are set. - FALSE (logic 0) if none of the tested bits are set.
Description	This function tests the required bits at **address** using the mask with the original value. If any of the bits are set, the function returns 1. If none of the bits are set, the function returns 0. **Note:** the bits_Set, bits_Clear, bits_Flip and bits_Test functions can only be used for user memory and will not work with system register variables.	
Example	`var myval,res;` `myval = 0x1234;` `res := bits_Test(myval, 0xFF00);` `print(res);` This example tests bits 8-15 in myval, if any bits are set, the result will be 1.	

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 arguments 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` 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);` `print(getsp());  // print the stack index` This example will print '1' 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);` `push(200);` `print(pop()+ pop());` 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 arguments 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;` `push(1234);` `push(5678);` `push(myvar);` 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 arguments 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;` `push(1234);` `push(5678);` `push(myvar);` `drop(2);` 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);` `push(10);` `push(50);` `push(50);` `push(0xFFFF);` `push(gfx_RectangleFilled); // push the function call address` `push(5); // push the argument count` `//~~~~~~~` `call();` 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 arguments 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;` `number := -100;` `myvar := ABS(number * 5);` 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;` `number1 := 33;` `number2 := 66;` `myvar := MIN(number1, number2);` 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;` `number1 := 33;` `number2 := 66;` `myvar := MAX(number1, number2);` This example returns 66 in variable **myvar**.	

2.4.4 SWAP(&var1, &var2)

Syntax	MAX(value1, value2);	
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;` `number1 := 33;` `number2 := 66;` `SWAP(number1, number2);` 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 arguments 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;` `angle := 133;` `myvar := SIN(angle);` 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 arguments 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;` `angle := 133;` `myvar := COS(angle);` 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);` `print(RAND(),", ",RAND());` This example will print **`3558, 1960`** 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 arguments 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);` `print(RAND(),", ",RAND());` This example will print **`30129, 27266`** to the display.	

2.4.9 SQRT(number)

Syntax	**SQRT(number);**	
Arguments	**number**	
	number	Specifies the positive number for the SQRT function.
	The arguments 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;` `myvar := SQRT(26000);` 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. It can be used with the shift operations, addition, subtraction, multiplication and modulus operations.	
Example	`var loWord, hiWord;` `loWord := 0x2710 * 0x2710;  // (10000 * 10000 in hex format)` `hiWord := OVF();` `print ("0x", [HEX] hiWord,  [HEX] loWord);` This example will print **0x05F5E100** 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);` 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 arguments 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;` `v := 0x39;` `putch(v);  // print the number 9 to the current display location` `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. **Note:** **putstr** is more efficient that **print** for printing single strings. The output of **putstr** can be redirected to the communications port, the media, or memory using the **to(...);** function. A string constant is automatically terminated with a zero. A string in a data statement is not automatically terminated with a zero. 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"
```

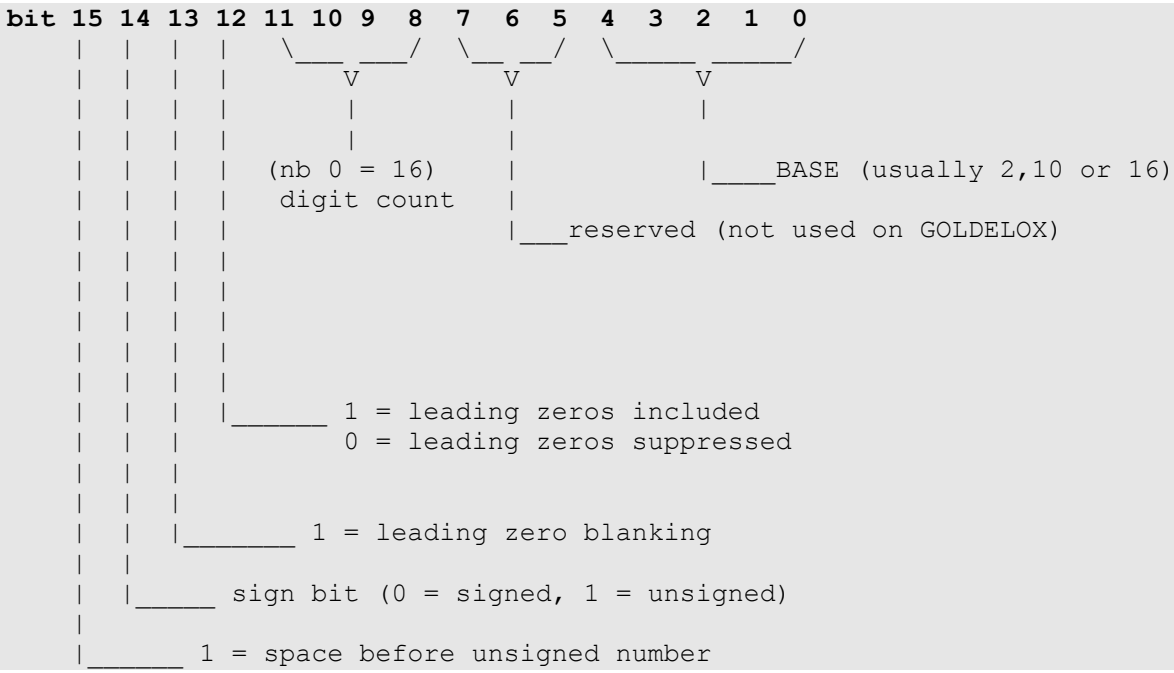
```
    byte wed "Wednesday\n\0"
    byte thu "Thursday\n\0"
    byte fri "Friday\n\0"
    byte sat "Saturday\n\0"
#END

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



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	**nothing**	
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. (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 (0 = system font)	See note #5
3	TEXT_WIDTH	Set the text width multiplier (note #6)	1 to 16 (note #7)
4	TEXT_HEIGHT	Set the text height multiplier (note #6)	1 to 16 (note #7)
5	TEXT_XGAP	Set the pixel gap between characters	0 to n (note #8)
6	TEXT_YGAP	Set the pixel gap between lines	0 to n (note #8)
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 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 (bits can be combined by using logical 'or' of bits) nb:- bits 0-3 and 8-15 are reserved	16 or BOLD 32 or ITALIC 64 or INVERSE 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 (0 = system font)	See note #5
txt_Width(multiplier)	Set the text width multiplier (note #6)	1 to 16 (note #7)
txt_Height(multiplier)	Set the text height multiplier (note #6)	1 to 16 (note #7)
txt_Xgap(pixelcount)	Set the pixel gap between characters	0 to n (note #8)
txt_Ygap(pixelcount)	Set the pixel gap between lines	0 to n (note #8)
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 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 (bits can be combined by using logical 'OR' of bits) nb:- bits 0-3 and 8-15 are reserved	16 or BOLD 32 or ITALIC 64 or INVERSE 128 or UNDERLINED

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_IncX()
- gfx_IncY()
- gfx_LineTo(xpos, ypos)
- gfx_LineRel(xpos, ypos)
- gfx_BoxTo(x2, y2)
- gfx_SetClipRegion()
- gfx_ClipWindow(x1, y1, x2, y2)
- gfx_FocusWindow()
- gfx_Set(function, value)

  gfx_Set shortcuts:

  - gfx_PenSize(mode)
  - gfx_BGcolour(colour)
  - gfx_ObjectColour(colour)
  - gfx_Clipping(mode)
  - gfx_FrameDelay(delay)
  - gfx_ScreenMode(delay)
  - 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);` `gfx_Cls();` 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. 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` `gfx_Circle(50,50,30, 0x001F);` 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. The outline colour can be specified with **gfx_OutlineColour(...).** If **OUTLINE_COLOUR** is set to 0, no outline is drawn. NB:- The **PEN_SIZE** is ignored, the circle is always drawn **SOLID.**	
Example	`gfx_OutlineColour(0xFFE0);` `gfx_CircleFilled(25,25,10, 0xF800);` 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);` 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);` 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);` 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. 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` `gfx_Rectangle(10, 10, 30, 30, 0x07E0);` 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. 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. NB:- The **PEN_SIZE** is ignored, the rectangle is always drawn **SOLID.**	
Example	`gfx_OutlineColour(0xFFE0);` `gfx_RectangleFilled(30,30,80,80, 0xF800);` 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);` 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);` `gfx_ObjectColour(0xF800);` `gfx_Dot();` 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 arguments 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. **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` `// and OBJECT_COLOUR is WHITE` `gfx_MoveTo(50,50);` `gfx_Bullet(5);` 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, &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;` `gfx_OrbitInit(&targetX, &targetY);` 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);` 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);` `r := gfx_GetPixel(20, 20);` `print(r);` 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);` `gfx_Dot();` 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);` `gfx_MoveRel(-5, -3);` `gfx_Dot();` This example draws a pixel using the current object colour at x=5, y=17	

2.6.21 gfx_IncX()

Syntax	gfx_IncX();	
Arguments	none	
Returns	old_origin	
	old_origin	Returns the current X origin before the increment.
Description	Increment the current X origin by 1 pixel unit. The original value is returned before incrementing. The return value can be useful if a function requires the current point before insetting occurs.	
Example	var n; gfx_MoveTo(20,20); n := 96; while (n--) gfx_ObjectColour(n/3); gfx_Bullet(2); gfx_IncX(); wend This example draws a simple rounded vertical gradient.	

2.6.22 gfx_IncY()

Syntax	gfx_IncY();	
Arguments	none	
Returns	old_Yorigin	
	old_Yorigin	Returns the current Y origin before the increment.
Description	Increment the current Y origin by 1 pixel unit. The original value is returned before incrementing. The return value can be useful if a function requires the current point before insetting occurs.	
Example	`var n;` `gfx_MoveTo(20,20);` `n := 96;` `while (n--)` `  gfx_ObjectColour(n/3);` `  gfx_LineRel(20, 0);` `  gfx_IncY();` `wend` This example draws a simple horizontal gradient using lines.	

2.6.23 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);` `gfx_LineTo(60, 70);` 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.	

2.6.24 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);` `gfx_MoveTo(10, 20);` `gfx_LineRel(50, 50);` This example draws a tessellated line using the current object colour between 10,20 and 50,50. **Note:** that **gfx_LinePattern(0);** must be used after this to return line drawing to normal solid lines.	

2.6.25 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. **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.	

2.6.26 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.

2.6.27 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 := 50000;` `while(n--)` `  gfx_PutPixel(RAND()%100, RAND()%100, RAND());` `wend` `repeat forever` This example will draw 50000 random colour pixels, only the pixels within the clipping area will be visible	

2.6.28 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.

```
// example #2
func main()
    var pixels;
    putstr("Open the terminal\n");
    putstr("Type any key to start\n");
    while(serin() < 0);    // wait for key from terminal before start
    gfx_ClipWindow(40,40,44,44);   // within a small block on display
    pixels:=gfx_FocusWindow();     //focus GRAM and get pixel count
    disp_BlitPixelFill(0x4142, pixels); // fill the area, using ASCII
                                        // values so we can read easy
    disp_BlitPixelsToCOM();  // send all the pixel values to com port
    print("Done!");
    repeat forever
endfunc
```

This example fills a small screen area, then outputs each pixel of the selected area to the COM port.

2.6.29 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	**nothing**	
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**) nb:- pen size is set to **OUTLINE** for normal operation	0 or **SOLID** 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. The clipping points are set with **gfx_ClipWindow(...)**	0 or 1 (**ON** or **OFF**)
4	**TRANSPARENT_COLOUR**	Not implemented on GOLDELOX-GFX2	n/a
5	**TRANSPARENCY**	Not implemented on GOLDELOX-GFX2	n/a
6	**FRAME_DELAY**	Set the inter frame delay for **media_Video(...)**	0 to 255msec
7	**SCREEN_MODE**	Set the delay between character printing	(Default 0msec)
8	**OUTLINE_COLOUR**	Outline colour for rectangles and circles (set to 0 for no effect)	Colour, 0-65535
9	**CONTRAST**	Set contrast value, 0 = display off, 1-16 = contrast level (only available on Goldelox Engineering samples, must be implemented in users code for GOLDELOX-GFX2 with external initialisation tables, refer to individual display driver data sheets)	0 or **OFF** 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 1 bits for pixels off
11	**COLOUR_MODE**	Sets 8 or 16bit colour mode (only available on GOLDELOX Engineering samples, must be implemented	0 or **COLOUR16** 1 or **COLOUR8**

		in users code for GOLDELOX-GFX2 with external initialisation tables, refer to individual display driver data sheets)	
0	**PEN_SIZE**	Set the draw mode for gfx_LineTo, gfx_LineRel, gfx_Dot, gfx_Bullet and gfx_BoxTo (default mode is **OUTLINE**) nb:- pen size is set to **OUTLINE** for normal operation	0 or **SOLID** 1 or **OUTLINE**

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 **Note**: pen size is set to **OUTLINE** for normal operation (default).	0 or **SOLID** 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. The clipping points are set with **gfx_ClipWindow(...)**	0 or 1 (**ON** or **OFF**)
gfx_FrameDelay(delay)	Set the inter frame delay for **media_Video(...)**	0 to 255msec
gfx_ScreenMode(delay)	Set the delay in milliseconds between character printing	(Default 0msec)
gfx_OutlineColour(colour)	Outline colour for rectangles and circles. (set to 0 for no effect)	Colour 0-65535
gfx_Contrast(value)	Set contrast value, 0 = display off, 1-16 = contrast level. (only available on Goldelox Engineering samples, must be implemented in users code for GOLDELOX-GFX2 with external initialisation tables, refer to individual display driver data sheets)	0 or **OFF** 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 1 bits for pixels off
gfx_ColourMode(mode)	Sets 8 or 16bit colour mode (only available on Goldelox Engineering samples, must be implemented in users code for GOLDELOX-GFX2 with external initialisation tables, refer to individual display driver data sheets)	0 or **COLOUR16** 1 or **COLOUR8**

2.7 Display I/O Functions

These functions allow direct display access for fast blitting operations.

Summary of Functions in this section:

- disp_Init(initTable, stateMachine)
- disp_WriteControl(value)
- disp_WriteByte(value)
- disp_WriteWord(value)
- disp_ReadByte()
- disp_ReadWord()
- disp_BlitPixelFill(colour, count)
- disp_BlitPixelsToMedia()
- disp_BlitPixelsFromMedia(pixelcount)
- disp_SkipPixelsFromMedia(pixelcount)
- disp_BlitPixelsToCOM()
- disp_BlitPixelsFromCOM(mode)

2.7.1 disp_Init(initTable, stateMachine)

Syntax	**disp_Init(initTable, stateMachine);**	
Arguments	**initTable, stateMachine**	
	initTable	A reference to the device initialisation table which is stored as a data statement.
	stateMachine	A reference to the device state machine table which is stored as a data statement.
	The arguments can be a variable, array element, expression or constant	
Returns	**nothing**	
Description	The GOLDELOX-GFX2 needs to be aware of all the display registers and how to access them. The initialisation and the state machine tables are necessary to achieve this. Refer to the individual display data sheet available from the display manufacturer. **Note:** for hardware platform modules such as uOLED-96-G1(GFX), uOLED-128-G1(GFX), etc the disp_Init(,,) is not needed. The modules are factory set-up with their display specific configurations.	
Example		

```
//=============================================================
/ SD1339 Device Initialisation Procedure
//=============================================================

#DATA
byte initTable
// first 4 bytes of table hold
// display access information
   _DISPLAY_X_MAX,         // width-1
   _DISPLAY_X_MAX,         // height-1
   WRITE_GRAM,             // write access register
   WRITE_GRAM,             // read access register
   // now the display initialisation table
   0, DISPLAY_OFF,         // Display OFF
   1, REMAP_COLOUR_SETTINGS,  _65K_COLOURS, // Set Re-map/Color Depth
   1, DISPLAY_START_LINE,     0x00,
   1, DISPLAY_OFFSET,         0x80,
   1, DUTY_CYCLE, 0x7F,                 // Duty 127+1 (0x80)
   0, DISPLAY_NORMAL,                   // Normal display
   1, MASTER_CONFIGURE, 0x8E,           // Set Master Configuration
   1, CONTRAST_MASTER, 0x0F,            // Set master contrast
   3, CONTRAST_RGB, 0xFF, 0xFF, 0xFF, // Set contrast current
   1, SET_VCOMH,  0x1F,                 // Set VcomH
   1, POWERSAVE_MODE,   0x05,           // Power saving mode
   3, PRECHARGE_VOLTAGE_RGB,  0x1C, 0x1C, 0x1C, // Set pre-charge
                                                // voltage
   1, PHASE_PRECHARGE,  0x11,           // Set pre & dis_charge
   1, CLOCK_FREQUENCY,  0x80,           // clock & frequency (0xF0)
   0, SLEEP_MODE_OFF,                   // Display on
   2, SET_COLUMN_ADDRESS, 0x00, 0x7F,  // set full screen
```

```
   2, SET_ROW_ADDRESS,  0x00, 0x7F,
   0xFF
#END

//=============================================================
// GRAM access state machine for SSD1339 (on uOLED-128-G1(GFX))
//=============================================================
#DATA
byte stateMachine
    WRITE_CONTROL_CONSTANT, SET_COLUMN_ADDRESS,
    WRITE_DATA_BYTE, _VX1,
    WRITE_DATA_BYTE, _VX2,
    WRITE_CONTROL_CONSTANT, SET_ROW_ADDRESS,
    WRITE_DATA_BYTE, _VY1,
    WRITE_DATA_BYTE, _VY2,
    WRITE_EXIT
#END

func main()
    disp_Init(initTable, stateMachine);   // initialise the display
    txt_MoveCursor(0, 2);
    txt_Bold(1);
    txt_Italic(1);
    txt_Set(TEXT_COLOUR, WHITE);
    print("4D LABS");
    repeat forever
end
```

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 arguments 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.	
Example		

```
// a function to utilise the hardware circle draw function
// on a SD1339 display driver IC
#constant DRAW_CIRCLE 0x86
func myCircle(var x, var y, var r, var fillcolour, var linecolour)
    disp_WriteControl(DRAW_CIRCLE);  // Draw Circle command
    disp_WriteByte(x);               // set x1
    disp_WriteByte(y);               // set y1
    disp_WriteByte(r);               // set x2
    disp_WriteByte(linecolour>>8);   // set outline colour Hi byte
    disp_WriteByte(linecolour);      // set outline colour Lo byte
    disp_WriteByte(fillcolour>>8);   // set fill colour Hi byte
    disp_WriteByte(fillcolour);      // set fill colour Lo byte
endfunc
```

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 arguments 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.	
Example		

```
// a function to utilise the hardware circle draw function
// on a SD1339 display driver IC
#constant DRAW_CIRCLE 0x86
func myCircle(var x, var y, var r, var fillcolour, var linecolour)
    disp_WriteControl(DRAW_CIRCLE);  // Draw Circle command
    disp_WriteByte(x);               // set x1
    disp_WriteByte(y);               // set y1
    disp_WriteByte(r);               // set x2
    disp_WriteByte(linecolour>>8);   // set outline colour Hi byte
    disp_WriteByte(linecolour);      // set outline colour Lo byte
    disp_WriteByte(fillcolour>>8);   // set fill colour Hi byte
    disp_WriteByte(fillcolour);      // set fill colour Lo byte
endfunc
```

2.7.4 disp_WriteWord(value)

Syntax	**disp_WriteWord(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 arguments can be a variable, array element, expression or constant	
Returns	**nothing**	
Description	Sends a 16 bit value to the display bus. Since the GOLDELOX-GFX2 display data bus is 8bits wide, the HIGH byte is sent first followed by the LOW byte. 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.	
Example		

```
// a function to utilise the hardware circle draw function
// on a SD1339 display driver IC
#constant DRAW_CIRCLE 0x86
func myCircle(var x, var y, var r, var fillcolour, var linecolour)
    disp_WriteControl(DRAW_CIRCLE);  // Draw Circle command
    disp_WriteByte(x);               // set x1
    disp_WriteByte(y);               // set y1
    disp_WriteByte(r);               // set x2
    disp_WriteWord(linecolour);      // set outline colour
    disp_WriteWord(fillcolour);      // set fill colour
endfunc
```

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` `gfx_FocusWindow();  // focus GRAM` `pixel_Hi:= dispReadByte();  // read hi byte of first pixel` `pixel_Lo:= dispReadByte();  // read lo byte of first pixel`	

2.7.6 disp_ReadWord()

Syntax	**disp_ReadWord();**	
Arguments	**none**	
Returns	**value**	
	value	Returns the 16bit data that was read from the display.
Description	Reads a 16bit word 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` `gfx_FocusWindow();  // focus GRAM` `pixel := dispReadWord();  // read 1st pixel, 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` `count := gfx_FocusWindow();  // focus GRAM` `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 arguments 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	...	

```
media_SetAdd(0x0002, 0x3C00);    // point to required area of an image
disp_BlitPixelsFromMedia(20);    // write the next 20 pixels from
                                 // media to the current GRAM pointer.
```

```
...
```

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 arguments 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_BlitPixelsToCOM()

Syntax	**disp_BlitPixelsToCOM();**	
Arguments	**none**	
Returns	**pixelcount**	
	pixelcount	Returns the number of pixels that were written to the serial port.
Description	Write the selected GRAM area to the serial (COM) port.	
Example		

```
// After downloading this program, open the Workshop Terminal and
// type any key to start the pixel upload.
func main()
   var pixels;
   putstr("Open the terminal\n");
   putstr("Type any key to start\n");
   while(serin() < 0);              // wait for a key from terminal
                                    // before we start
   gfx_ClipWindow(40,40,44,44);     // within a small block on the
                                    // display
   pixels:=gfx_FocusWindow();       // focus GRAM and get pixel count
                                    // of area
   disp_BlitPixelFill(0x4142, pixels);// fill the area  using ASCII
                                       // values so we can read easily
   disp_BlitPixelsToCOM();          // write the pixels to the COM port
   print("Done!");
   repeat forever
endfunc
```

2.7.12 disp_BlitPixelsFromCOM(mode)

Syntax	**disp_BlitPixelFromCOM(mode);**	
Arguments	**mode**	
	mode	mode = 0 : specifies 16 bit pixels mode = pointer : specifies pointer to 16 element colour lookup table for each 4bit pixel value
	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		

```
// After downloading this program, open the Workshop Terminal and
// type 2 keys per pixel for 16bit colour mode. The colour will be
// determined by the ASCII values of the keys, it is only a simple
// test and you have very little control of what colour is actually
// displayed - it is simply a demo of disp_BlitPixelsFromCOM action.
// If all is good, you will see the GRAM area being filled with
// pixels.
// NB if using 8bit colour mode, the correct register in the display
// must be set to 8 bit mode, if you have done this correctly, you
// will notice that it only requires 1 key to write each pixel.
// If this is not done correctly, only half the gram area will be
// filled.

func main()
   gfx_ClipWindow(40,40,59,59);  // writing to a 40x40 block on the
                                 // display.
   gfx_FocusWindow();            // NB first focus is just so we can
                                 // get pixel count of area.
   print("Filling ",*IMG_PIXEL_COUNT," pixels");
   gfx_FocusWindow();
   disp_BlitPixelsFromCOM(0);    // get pixels from serial port,

   while(*IMG_PIXEL_COUNT);      // wait till all the pixels come in
   txt_MoveCursor(8,5);
   print("Done!");
   repeat forever
endfunc

//=================================================================

// the next example uses  disp_BlitPixelsFromCOM in 4bit CLUT mode
var CLUT1[16];
// If the argument to disp_BlitPixelsFromCOM(...) is non zero, it is
// expected to be a pointer to a 16 element colour lookup table in
// RAM.
// After downloading this program, open the Workshop Terminal and
// Each key typed will produce 2 pixels from the CLUT. The colour
// will be determined by the values in the CLUT, it is only a simple
```

```
// test and you have very little control of what colour is actually
// displayed - it is simply a demo of disp_BlitPixelsFromCOM action.
// If all is good, you will see the GRAM area being filled with
// pixels.

func main()
   // CLUT is set for monochrome mode, however
   // it can contain a colour set if required
   CLUT1[0] := 0x0000;     // BLACK
   CLUT1[1] := 0x1082;     // GRAY1
   CLUT1[2] := 0x2104;     // GRAY2
   CLUT1[3] := 0x3186;     // GRAY3
   CLUT1[4] := 0x4208;     // GRAY4
   CLUT1[5] := 0x5285;     // GRAY5
   CLUT1[6] := 0x630C;     // GRAY6
   CLUT1[7] := 0x738E;     // GRAY7
   CLUT1[8] := 0x8410;     // GRAY8
   CLUT1[9] := 0x9492;     // GRAY9
   CLUT1[10] := 0xA514;    // GRAY10
   CLUT1[11] := 0xB596;    // GRAY11
   CLUT1[12] := 0xC618;    // GRAY12
   CLUT1[13] := 0xD69A;    // GRAY13
   CLUT1[14] := 0xE71C;    // GRAY14
   CLUT1[15] := 0xF79E;    // ALMOST WHITE

   gfx_ClipWindow(40,40,59,59);   // writing to a 40x40 block on
                                  // the display.
   gfx_FocusWindow();    // NB first focus is just so we can get
                         // pixel count of area.
   print("Filling ",*IMG_PIXEL_COUNT," pixels");
   gfx_FocusWindow();
   disp_BlitPixelsFromCOM(CLUT1); // get pixels from COM port, 4 bit
                                  // CLUT mode mode
   while(*IMG_PIXEL_COUNT);
   txt_MoveCursor(8,5);
   print("Done!");
   repeat forever
endfunc
```

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-GFX2 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)

2.8.1 media_Init()

Syntax	**media_Init();**	
Arguments	**none**	
Returns	**result**	
	result	Returns: **1** if memory card is present and successfully initialised 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-GFX2 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);` 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);` 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;` `if(media_Init())` `  media_SetAdd(0, 510);` `  myword := media_ReadWord();` `  print([HEX4]myword);` `endif` `repeat forever` 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 arguments 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(...);** **Note:** Due to design constraints on the GOLDELOX-GFX2, 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 arguments 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(...);** **Note:** Due to design constraints on the GOLDELOX-GFX2, 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. **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. **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	See below	

```
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
```

2.9 Flash Memory Chip Functions

The functions in this section only apply to serial SPI (NAND) flash devices interfaced to the GOLDELOX-GFX2 SPI port.

Summary of Functions in this section:

- flash_SIG()
- flash_ID()
- flash_BulkErase()
- flash_BlockErase(blockAddress)

2.9.1 flash_SIG()

Syntax	**flash_SIG();**	
Arguments	**none**	
Returns	**signature**	
	signature	Release from Deep Power-down, and Read Electronic Signature. Only the low order byte is valid, the upper byte is ignored.
Description	If a FLASH storage device is connected to the SPI port, and has been correctly initialised with the **spi_Init(...)** function, the Electronic Signature of the device can be read using this function. The only devices supported so far on the GOLDELOX-GFX2 are the M25Pxx range of devices which are 512Kbit to 32Mbit (2M x 8) Serial Flash Memory.	

2.9.2 flash_ID()

Syntax	**flash_ID();**	
Arguments	**none**	
Returns	**type_capacity**	
	type_capacity	Reads the memory type and capacity from the serial FLASH device. Hi byte contains type, and low byte contains capacity. Refer to the device data sheet for further information.
Description	If a FLASH storage device is connected to the SPI port, and has been correctly initialised with the **spi_Init(...)** function, the memory type and capacity from the flash device can be read using this function. The only devices supported so far on the GOLDELOX-GFX2 are the M25Pxx range of devices which are 512Kbit to 32Mbit (2M x 8) Serial Flash Memory.	

2.9.3 flash_BulkErase()

Syntax	flash_ID();
Arguments	**none**
Returns	**nothing**
	Erases the entire flash media device. The function returns no value, and the operation can take up to 80 seconds depending on the size of the flash device.
Description	If a FLASH storage device is connected to the SPI port, and has been correctly initialised with the **spi_Init(...)** function, the FLASH device can be completely erased using this function. The only devices supported so far on the GOLDELOX-GFX2 are the M25Pxx range of devices which are 512Kbit to 32Mbit (2M x 8) Serial Flash Memory.

2.9.4 flash_BlockErase(blockAddress)

Syntax	flash_BlockErase(blockAddress);	
Arguments	**blockAddress**	
	blockAddress	The address of the 64k FLASH block to be erased.
Returns	**result**	
	result	Erases the required block in a FLASH media device. The function returns no value, and the operation can take up to 3 milliseconds.
Description	If a FLASH storage device is connected to the SPI port, and has been correctly initialised with the **spi_Init(...)** function, the FLASH block can be erased using this function. The only devices supported so far on the GOLDELOX-GFX2 are the M25Pxx range of devices which are 512Kbit to 32Mbit (2M x 8) Serial Flash Memory. E.g. there are 32 x 64K blocks on a 2Mb flash device.	

2.10 SPI Control Functions

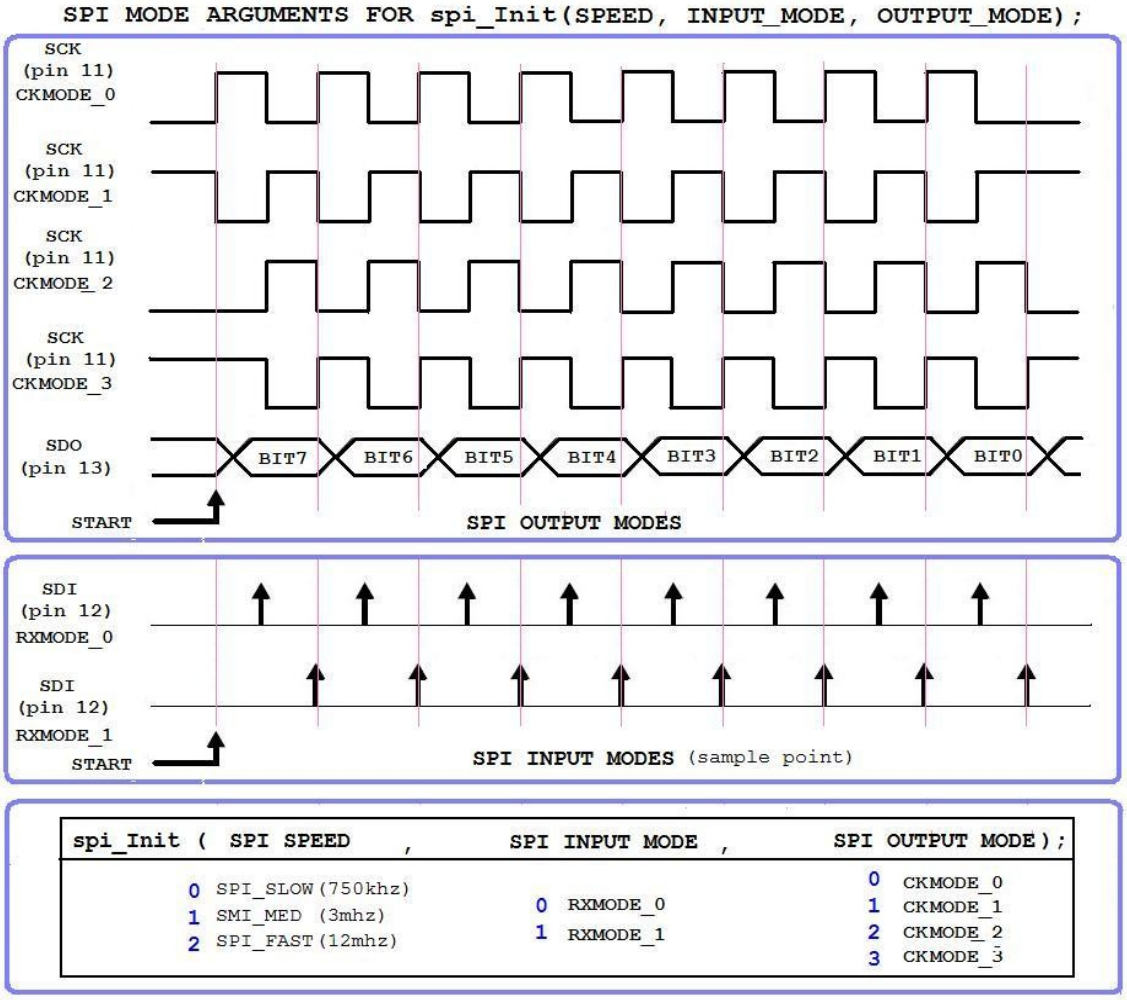
The SPI functions in this section apply to any general purpose SPI device.

Summary of Functions in this section:

- spi_Init(speed, input_mode, output_mode)
- spi_Read()
- spi_Write(byte)
- spi_Disable()

2.10.1 spi_Init(speed, input_mode, output_mode)

Syntax	spi_Init(speed, input_mode, output_mode);	
Arguments	**speed, input_mode, output_mode**	
	speed	Sets the speed of the SPI port.
	input_mode	Sets the input mode of the SPI port. See diagram below.
	output_mode	Sets the output mode of the SPI port. See diagram below.
	The arguments can be a variable, array element, expression or constant	
Returns	**nothing**	
Description	Sets up the GOLDELOX-GFX2 SPI port to communicate with SPI devices. **Note:** The SPI functions in this section are not necessary when using the memory card or serial flash chips interfaced to the SPI port. The SPI functions in this section are relevant to those devices other than the memory card and the serial flash chip used for media access.	



2.10.2 spi_Read()

Syntax	spi_Read();	
Arguments	none	
Returns	byte	
	byte	Returns a single data byte from the SPI device.
Description	This function allows a raw unadorned byte read from the SPI device. **Note:** The Chip Select line (SDCS) is lowered automatically.	

2.10.3 spi_Write(byte)

Syntax	**spi_Write(byte);**	
Arguments	**byte**	
	byte	specifies the data byte to be sent to the SPI device.
	The arguments can be a variable, array element, expression or constant	
Returns	**nothing**	
Description	This function allows a raw unadorned byte write to the SPI device. **Note:** The Chip Select line (SDCS) is lowered automatically.	

2.10.4 spi_Disable()

Syntax	spi_Disable();
Arguments	none
Returns	nothing
Description	This function raises the Chip Select (SDCS) line of the SPI device, disabling it from further activity. The CS line will be automatically lowered next time the SPI functions spi_Read() or spi_Write(...) are used, and also by action of any of the media_ functions.

2.11 Serial (UART) Communications Functions

Summary of Functions in this section:

- serin()
- serout(char)
- setbaud(rate)
- com_AutoBaud(timeout)
- com_Init(buffer, buffsize, qualifier)
- com_Reset()
- com_Count()
- com_Full()
- com_Error()
- com_Sync()
- com_Checksum()
- com_PacketSize()

2.11.1 serin()

Syntax	**serin();**	
Arguments	**none**	
Returns	**char**	
	char	Returns: **-1** if no character is available Returns: **-2** if a framing error or over-run has occurred (auto cleared) Returns: positive value **0 to 255** for a valid character received
Description	Receives a character from the Serial Port COM0. The transmission format is: **No Parity, 1 Stop Bit, 8 Data Bits** (N,8,1). 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;` `char := serin();  // test the com port` `if (char >= 0)  // if a valid character is received` `  process(char);  // process the character` `endif`	

2.11.2 serout(char)

Syntax	**serout(char);**	
Arguments	**char**	
	char	specifies the data byte to be sent to the serial port.
	The arguments 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: **No Parity, 1 Stop Bit, 8 Data Bits** (N,8,1). 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		

2.11.3 setbaud(rate)

Syntax	setbaud(rate);			
Arguments	rate			
	rate	specifies the baud rate divisor value or pre-defined constant		
	The arguments 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. 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_AutoBaud(timeout)

Syntax	**com_AutoBaud(timeout);**	
Arguments	**timeout**	
	timeout	Sets the timeout delay for autobaud detection.
	The arguments can be a variable, array element, expression or constant	
Returns	**status**	
	status	Returns the divisor value selected for the baud rate generator, else returns 0.
Description	The com_AutoBaud function expects to receive an ascii 'U' (0x55) within a pre-determined time. If the function is successful, the COM port is configured to the closest speed possible, and the selected baud rate value is returned.	
Example	`while (br:=com_AutoBaud(500))  // if we receive a 'U' ok` `  doMyComms();  // now connected at br baud rate` `endif`	

2.11.5 com_Init(buffer, bufsize, qualifier)

Syntax	**com_Init(buffer, bufsize, qualifier);**	
Arguments	**buffer, bufsize, 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).
	qualifier	specifies the qualifying character that must be received to initiate serial data reception and buffer write. A zero (0x00) 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. The service also transparently keeps a checksum (see the **com_Checksum()** function) which can be employed if required for robust error checking. **MODES OF OPERATION** • **No *qualifier* – simple ring buffer (aka circular queue)** If the ***qualifier*** is set to zero, the ***buffer*** is continually active 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. 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**

  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.

- **Variable packet length**

  If the ***bufsize*** argument is set to zero, the first byte received (or the 2nd byte if a ***qualifier*** is employed) sets the count of characters that are to be received before the COM_FULL flag (which can be read with the **com_Full()** function) becomes set. This allows a host to send variable length packets, which will only alert the application that the packet is ready after the correct number of characters has been received. The number of bytes to be expected can be read using the **com_PacketSize()** function, which will indicate the packet size**.** In this mode, it is wise to make the buffer as large as possible due to the fact that if the 'size' parameter sent by the host is corrupted, more characters than expected (up to 255) can be receive inadvertently, crashing into any other program variables above the array.

Notes:

- Transparent to normal operation, a check summing system is operating. If the host sends one extra character (usually at the end of the packet) which is the negated value of the addition of all the previous characters in the packet , the checksum (which can be read with the **com_Checksum()** function) should read zero. **com_Checksum()** will retain the most recent value until **com_Init(..)** is called again to reset the buffer system. Note that the checksum is only valid after the **com_Full()** function reports a buffer full situation (ie the packet is fully received).

- **com_PacketSize()** will indicate how large the packet is ONLY after the packet reception has started. Although it is usually not required to know the packet size until the packet has actually been read, if it is a requirement, the count is available as soon as **com_Count()** becomes non zero.

Example

```
//================================================
// Example #1 - no qualifier
```

```
// use the Workshop Terminal to test this example
// note that if 7 characters are exceeded, no more
// characters will be accepted as there is no action
// to take care of the com_Full situation
//=================================================
var combuf[10];             // a buffer for up to 20 characters
putstr("Default 115.2kb");
com_Init(combuf, 7, 0);     // initialize small circular queue of 7
                            // bytes, no qualifier
repeat
    if(com_Count())         // if there is a character available
        serout(serin());    // echo it back to host
    endif
    txt_MoveCursor(2,0);
    print("\ncom_Error ",[DEC2ZB] com_Error()); // 1 if error
    print("\ncom_Count ",[DEC2ZB] com_Count()); // show current count
    print("\ncom_Full  ",[DEC2ZB] com_Full());  // 1 if full
    pause(1000);  // a delay to slow things up
forever



//=================================================
// Example #2 - no qualifier
// use the Workshop Terminal to test this example
// note that if 7 characters are exceeded, the
// com_Full situation occurs, but is reset
// once all the pending characters are read
//=================================================
var combuf[10];             // a buffer for up to 20 characters
putstr("Default 115.2kb");
com_Init(combuf, 7, 0);     // initialize circular queue of 7 bytes,
                            // no qualifier
repeat
    if(com_Count())         // if there is a character available
        serout(serin());    // echo it back to host
    endif
    txt_MoveCursor(2,0);
    print("\ncom_Error ",[DEC2ZB] com_Error()); // 1 if error
    print("\ncom_Count ",[DEC2ZB] com_Count()); // show current count
    print("\ncom_Full  ",[DEC2ZB] com_Full());  // 1 if full
    pause(1000); // a delay to slow things up

    // if the buffer overflowed, and we have read
    // all the characters, then reset the buffer
    if (com_Full() & (com_Count() == 0)) com_Init(combuf, 7, '0');
forever


//=================================================
// Example #3 - using qualifier (a colon character)
// use the Workshop Terminal to test this example
// note that once the qualifier is received, if 7
// characters are exceeded, the buffer is reset
// once all the pending characters are read
//=================================================
var combuf[10];             // a buffer for up to 20 characters
putstr("Default 115.2kb");
```

```
com_Init(combuf, 7, ':');   // initialize circular queue of 7 bytes,
                            // ':' as qualifier
repeat
    if(com_Count())         // if there is a character available
        serout(serin());    // echo it back to host
    endif
    txt_MoveCursor(2,0);
    print("\ncom_Sync  ",[DEC2ZB] com_Sync());  // 1 if qualified
    print("\ncom_Error ",[DEC2ZB] com_Error()); // 1 if error
    print("\ncom_Count ",[DEC2ZB] com_Count()); // show current count
    print("\ncom_Full  ",[DEC2ZB] com_Full());  // 1 if full
    pause(1000); // a delay to slow things up

    // if the buffer overflowed, if we have read
    // all the characters, then reset the buffer
    if (com_Full() & (com_Count() == 0)) com_Init(combuf, 5, ':');
forever



//=================================================
// Example #4 - using qualified packet
// use the Workshop Terminal to test this example
// note that nothing happens until the qualifier
// followed by 10 characters is received. Then an
// acknowledgement is issued to the host, and the
// buffer is reset
//=================================================
var combuf[10], chr;          // a buffer for up to 20 characters
putstr("Default 115.2kb");
com_Init(combuf, 10, ':');    // init buffer 10 bytes to receive

repeat
    repeat
        txt_MoveCursor(2,0);
        print("\ncom_Sync  ",[DEC2ZB] com_Sync());  // 1 if qualified
        print("\ncom_Error ",[DEC2ZB] com_Error()); // 1 if error
        print("\ncom_Count ",[DEC2ZB] com_Count()); // show count
        print("\ncom_Full  ",[DEC2ZB] com_Full());  // 1 if full
        pause(1000);      // a delay to slow things up
    until(com_Full());   // just loop until buffer is full

    // buffer is full, echo the characters
    while (chr:=serin()) >=0) serout(chr); // echo back characters
    to(COM0); print(" OK\n");              // send an acknowledgement
    com_Init(combuf, 10, ':');  // re-init buffer 10 bytes to receive
forever // do it all again



//=================================================
// Example #5 - using qualified variable length packet
// use the Workshop Terminal to test this example
// NB:- to make the example possible when just using
// a terminal to emulate a packet, the 'space bar'
// (ascii 32) is used to set the size of the packet
// to 32 characters, so you must send the':' qualifier
// then press the space bar (you will then see '32'
// for the packet size) then type 32 characters to
```

```
// complete the action. Under normal circumstances,
// the host will send whatever packet size is required.
// Note that nothing happens until the qualifier ':'
// followed by the space bar (to set the packet size),
// then the 32 characters are received. After the
// packet is received, the acknowledgement is issued
// to the host, and the buffer is reset.
// This example also shows the running checksum
// calculation.
//================================================

putstr("Default 115.2kb");
repeat
    com_Init(combuf, 0, ':');  // init. buffer 10 bytes to receive
    repeat
      txt_MoveCursor(2,0);
      print("\ncom_Sync      ",[DEC2ZB] com_Sync());    // 1 if
                                                       // qualified
      print("\ncom_Error      ",[DEC2ZB] com_Error());  // 1 if error
      print("\ncom_PacketSize ",[DEC2ZB] com_PacketSize());
      print("\ncom_Count      ",[DEC2ZB] com_Count());  // show count
      print("\ncom_Checksum   ",[HEX2ZB] com_Checksum()); // checksum
      print("\ncom_Full       ",[DEC2ZB] com_Full());    // 1 if full
      pause(1000);            // a delay to slow things up
    until(com_Full());        // just loop until buffer is full

    // buffer is full, echo the characters
    while ( (chr:=serin()) >= 0 ) serout(chr); // echo back the chars
    to(COM0); print(" OK\n");   // send a simple acknowledgement
forever    // do it all again
```

2.11.6 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`

2.11.7 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`	

2.11.8 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.	
Example	`if(com_Full() & (com_Count() == 0))` `  com_Init(mybuf, 30, 0);  // buffer full, recovery` `endif`	

2.11.9 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,` `  resetMySystem();  // take corrective action` `endif`	

2.11.10 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 initialized 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`	

2.11.11 com_Checksum()

Syntax	**com_Checksum();**	
Arguments	**none**	
Returns	**status**	
	status	Returns **0** if checksum has been computed correctly.
Description	Transparent to normal operation, a check summing system is operating. If the host sends one extra character as part of the packet (usually added at the end of the packet) which is the negated value of the addition of all the previous characters in the packet. Once the **com_Full()** function reports a buffer full situation (ie the packet is fully received) , the checksum can be read, and should read zero if the packet is not corrupted.	
Example	`if(!com_Checksum())  // if checksum is ok` `  processMyPacket();  // continue` `else` `  … do recovery action` `endif`	

2.11.12 com_PacketSize()

Syntax	**com_PacketSize();**	
Arguments	**none**	
Returns	**size**	
	size	Returns the size of a packet if in variable packet length mode, or just the size of the serial buffer if not variable packet length mode.
Description	**com_PacketSize()** will indicate how large the packet is ONLY after the packet reception has started. Although it is usually not required to know the packet size until the packet has actually been read, if it is a requirement, the count is available as soon as **com_Count()** becomes non zero. If not in variable packet length mode, **com_PacketSize()** just returns the size of the specified buffer.	
Example	`If (!com_Count())` `  print("Waiting....");` `else` `  print(com_PacketSize() - com_Count()), " bytes to go");  //` `endif`	

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. **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 arguments 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: http://www.activexperts.com/xmstoolkit/sms/rtttl/ and http://en.wikipedia.org/wiki/Ring_Tone_Transfer_Language 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. **The 4DGL implementation:** • The "**b=nnn**" in 4DGL does not represent "beats per minute" (bpm), it represents "milliseconds per hemidemisemiquaver". 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. • 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). (if a string is passed directly as a parameter, the zero (0x00) is automatically appended by the compiler as per normal strings). • The original RTTTL format is a string divided into three sections: **name, default value, data.** The 4DGL implementation does not have the "name" section - this would be just a waste of space. • 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
```

```
   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;

   // pin_Set(SOUND, PIN_1);          // sound on default pin
   // pin_Set(SOUND, PIN_2);

   // 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 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 arguments 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` `  pause(30)  // slow down the loop` `else` `  ...`	

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.** **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. **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 : `#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. **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: `#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-GFX2 EVE System Registers Memory Map

The following tables outline in detail the GOLDELOX-GFX2 system registers and flags.

Table 3.1: BYTE-Size Registers Memory Map

LABEL	ADDRESS		USAGE	SIZE	*NOTES
	DEC	HEX			
VX1	128	0x80	display hardware GRAM x1 pos	BYTE	SYSTEM (R/O)
VY1	129	0x81	display hardware GRAM y1 pos	BYTE	SYSTEM (R/O)
VX2	130	0x82	display hardware GRAM x2 pos	BYTE	SYSTEM (R/O)
VY2	131	0x83	display hardware GRAM y2 pos	BYTE	SYSTEM (R/O)
SYS_X_MAX	132	0x84	display hardware X res-1	BYTE	SYSTEM (R/O)
SYS_Y_MAX	133	0x85	display hardware Y res-1	BYTE	SYSTEM (R/O)
WRITE_GRAM_REG	134	0x86	display GRAM write address	BYTE	SYSTEM (R/O)
READ_GRAM_REG	135	0x87	display GRAM read address	BYTE	SYSTEM (R/O)
IMAGE_WIDTH	136	0x88	loaded image/animation width	BYTE	SYSTEM (R/O)
IMAGE_HEIGHT	137	0x89	loaded image/animation height	BYTE	SYSTEM (R/O)
IMAGE_DELAY	138	0x8A	frame delay (if animation)	BYTE	USER
IMAGE_MODE	139	0x8B	image/animation colour mode	BYTE	SYSTEM (R/O)
CLIP_LEFT_POS	140	0x8C	left clipping point setting	BYTE	USER
CLIP_TOP_POS	141	0x8D	top clipping point setting	BYTE	USER
CLIP_RIGHT_POS	142	0x8E	right clipping point setting	BYTE	USER
CLIP_BOTTOM_POS	143	0x8F	bottom clipping point setting	BYTE	USER
CLIP_LEFT	144	0x90	left clipping point active	BYTE	USER
CLIP_TOP	145	0x91	top clipping point active	BYTE	USER
CLIP_RIGHT	146	0x92	right clipping point active	BYTE	USER
CLIP_BOTTOM	147	0x93	bottom clipping point active	BYTE	USER
FONT_TYPE	148	0x94	0 = fixed, 1 = proportional	BYTE	SYSTEM (R/O)
FONT_MAX	149	0x95	number of chars in font set	BYTE	SYSTEM (R/O)
FONT_OFFSET	150	0x96	ASCII offset (usually 0x20)	BYTE	SYSTEM (R/O)
FONT_WIDTH	151	0x97	width of font (pixel units)	BYTE	SYSTEM (R/O)
FONT_HEIGHT	152	0x98	height of font (pixel units)	BYTE	SYSTEM (R/O)
TEXT_XMAG	153	0x99	text width magnification	BYTE	USER
TEXT_YMAG	154	0x9A	text height magnification	BYTE	USER
TEXT_MARGIN	155	0x9B	text place holder for CR	BYTE	SYSTEM (R/O)

TEXT_DELAY	156	0x9C	text delay effect (0-255msec)	BYTE	USER
TEXT_X_GAP	157	0x9D	X pixel gap between chars	BYTE	USER
TEXT_Y_GAP	158	0x9E	Y pixel gap between chars	BYTE	USER
GFX_XMAX	159	0x9F	width of current orientation	BYTE	SYSTEM (R/O)
GFX_YMAX	160	0xA0	height of current orientation	BYTE	SYSTEM (R/O)
GFX_SCREENMODE	161	0xA1	Current screen mode (0-3)	BYTE	SYSTEM (R/O)
reserved	162-165	0xA2-0xA5	reserved	BYTE	SYSTEM (R/O)

*** NOTES:**	
SYSTEM	SYSTEM registers are maintained by internal system functions and should not be written to. They should only ever be read. **DO NOT WRITE to these registers.**
USER	USER registers are read/write (R/W) registers used to alter the system behaviour. Refer to the individual functions for information on the interaction with these registers.
These registers are accessible with **peekB** and **pokeB** functions.	

Table 3.2: WORD-Size Registers Memory Map

LABEL	ADDRESS		USAGE	SIZE	*NOTES
	DEC	HEX			
SYS_OVERFLOW	83	0x53	16bit overflow register	WORD	USER
SYS_COLOUR	84	0x54	internal variable for colour	WORD	SYSTEM
SYS_RETVAL	85	0x55	return value of last function	WORD	SYSTEM
GFX_BACK_COLOUR	86	0x56	screen background colour	WORD	USER
GFX_OBJECT_COLOUR	87	0x57	graphics object colour	WORD	USER
GFX_TEXT_COLOUR	88	0x58	text foreground colour	WORD	USER
GFX_TEXT_BGCOLOUR	89	0x59	text background colour	WORD	USER
GFX_OUTLINE_COLOUR	90	0x5A	circle/rectangle outline	WORD	USER
GFX_LINE_PATTERN	91	0x5B	line draw tessellation	WORD	USER
IMG_PIXEL_COUNT	92	0x5C	count of pixels in image	WORD	SYSTEM
IMG_FRAME_COUNT	93	0x5D	count of frames in animation	WORD	SYSTEM
MEDIA_HEAD	94	0x5E	media sector head position	WORD	SYSTEM
SYS_OUTSTREAM	95	0x5F	Output stream handle	WORD	SYSTEM
GFX_LEFT	96	0x60	image left real point	WORD	SYSTEM
GFX_TOP	97	0x61	image top real point	WORD	SYSTEM
GFX_RIGHT	98	0x62	image right real point	WORD	SYSTEM
GFX_BOTTOM	99	0x63	image bottom real point	WORD	SYSTEM
GFX_X1	100	0x64	image left clipped point	WORD	SYSTEM
GFX_Y1	101	0x65	image top clipped point	WORD	SYSTEM
GFX_X2	102	0x66	image right clipped point	WORD	SYSTEM
GFX_Y2	103	0x67	image bottom clipped point	WORD	SYSTEM
GFX_X_ORG	104	0x68	current X origin	WORD	USER
GFX_Y_ORG	105	0x69	current Y origin	WORD	USER
RANDOM_LO	106	0x6A	random generator LO word	WORD	SYSTEM
RANDOM_HI	107	0x6B	random generator HI word	WORD	SYSTEM
MEDIA_ADDR_LO	108	0x6C	media byte address LO	WORD	SYSTEM
MEDIA_ADDR_HI	109	0x6D	media byte address HI	WORD	SYSTEM
SECTOR_ADDR_LO	110	0x6E	media sector address LO	WORD	SYSTEM
SECTOR_ADDR_HI	111	0x6F	media sector address HI	WORD	SYSTEM
SYSTEM_TIMER_LO	112	0x70	1msec system timer LO word	WORD	USER
SYSTEM_TIMER_HI	113	0x71	1msec system timer HI word	WORD	USER

TIMER0	114	0x72	1msec user timer 0	WORD	USER
TIMER1	115	0x73	1msec user timer 1	WORD	USER
TIMER2	116	0x74	1msec user timer 2	WORD	USER
TIMER3	117	0x75	1msec user timer 3	WORD	USER
INCVAL	118	0x76	predec/preinc/postdec/postinc addend	WORD	USER
TEMP_MEDIA_ADDRLO	119	0x77	temporary media address LO	WORD	SYSTEM
TEMP_MEDIA_ADDRHI	120	0x78	temporary media address HI	WORD	SYSTEM
GFX_TRANSPARENTCOLOUR	121	0x79	Image transparency colour	WORD	USER
GFX_STRINGMETRIX	122	0x7A	Low byte = string width High byte = string height	WORD	SYSTEM
GFX_TEMPSTORE1	123	0x7B	Low byte = last character printed High byte = video frame timer over-ride	WORD	SYSTEM
reserved	124	0x7C	reserved	WORD	SYSTEM
reserved	125	0x7D	reserved	WORD	SYSTEM
SYS_FLAGS1	126	0x7E	system control flags word 0	WORD	FLAGS
SYS_FLAGS2	127	0x7F	system control flags word 1	WORD	FLAGS
USR_SP	128	0x80	User defined stack pointer	WORD	USERSTACK
USR_MEM	129	0x81	255 user variables / array(s)	WORD	MEMORY
SYS_STACK	384	0x180	128 level EVE machine stack	WORD	SYSTEMSTACK

*** NOTES:**	
SYSTEM	SYSTEM registers are maintained by internal system functions and should not be written to. They should only ever be read. **DO NOT WRITE to these registers.**
USER	USER registers are read/write (R/W) registers used to alter the system behaviour. Refer to the individual functions for information on the interaction with these registers.
USERSTACK	Used by the debugging and system extension utilities
MEMORY	255 word size variables for users program
STACK	128 word EVE system stack (STACK grows upwards)
FLAGS	FLAGS are a mixture of bits that are either maintained by internal system functions or set / cleared by various system functions. Refer to the FLAGS Register Bit Map table, and individual functions for further details.
These registers are accessible with **peekW** and **pokeW** functions.	

Table 3.3: FLAG Registers Bit Map

REGISTER	ADDRESS		NAME	USAGE	*NOTES	VALUE
	DEC	HEX				
SYS_FLAGS1	**126**	**0x7E**	*** denotes auto reset**			
	Bit 0		_STREAMLOCK	Used internally	SYSTEM	0x0001
	Bit 1		_PENSIZE	Object, 0 = solid, 1 = outline	SYSTEM	0x0002
	Bit 2		_OPACITY	Text, 0 = transparent, 1 = opaque	SYSTEM	0x0004
	Bit 3		_OUTLINED	box/circle outline 0 = off, 1 = on	SYSTEM	0x0008
	Bit 4		_BOLD	* text, 0 = normal, 1 = bold	SYSTEM	0x0010
	Bit 5		_ITALIC	* Text, 0 = normal, 1 = italic	SYSTEM	0x0020
	Bit 6		_INVERSE	* Text, 0 = normal, 1 = inverse	SYSTEM	0x0040
	Bit 7		_UNDERLINED	* Text, 0 = normal, 1 = underlined	SYSTEM	0x0080
	Bit 8		_CLIPPING	0 = clipping off, 1 = clipping on	SYSTEM	0x0100
	Bit 9		_STRMODE	Used internally	SYSTEM	0x0200
	Bit 10		_SERMODE	Used internally	SYSTEM	0x0400
	Bit 11		_TXTMODE	Used internally	SYSTEM	0x0800
	Bit 12		_MEDIAMODE	Used internally	SYSTEM	0x1000
	Bit 13		_PATTERNED	Used internally	SYSTEM	0x2000
	Bit 14		_COLOUR8	Display mode, 0 = 16bit, 1 = 8bit	SYSTEM	0x4000
	Bit 15		_MEDIAFONT	0 = internal font, 1 = media font	SYSTEM	0x8000
SYS_FLAGS2	**127**	**0x7F**				
	Bit 0		_MEDIA_INSTALLED	SD/SDHC or FLASH is detected/active	SYSTEM	0x0001
	Bit 1		_MEDIA_TYPE	0 = SD/SDHC, 1 = FLASH chip	SYSTEM	0x0002
	Bit 2		_MEDIA_READ	1 = MEDIA read in progress	SYSTEM	0x0004
	Bit 3		_MEDIA_WRITE	1 = MEDIA write in progress	SYSTEM	0x0008
	Bit 4		_OW_PIN	0 = IO1, 1 = IO2 (Dallas OW Pin)	SYSTEM	0x0010
	Bit 5		_PTR_TYPE	Used internally	SYSTEM	0x0020
	Bit 6		_TEMP1	Used internally	SYSTEM	0x0040
	Bit 7		_TEMP2	Used internally	SYSTEM	0x0080
	Bit 8		_RUNMODE	1 = running pcode from media	SYSTEM	0x0100
	Bit 9		_SIGNED	0 = number printed '-' prepend	SYSTEM	0x0200
	Bit 10		_RUNFLAG	1 = EVE processor is running	SYSTEM	0x0400
	Bit 11		_SINGLESTEP	1 = set breakpoint for debugger	SYSTEM	0x0800
	Bit 12		_COMMINT	1 = buffered coms active	SYSTEM	0x1000

	Bit 13	_DUMMY16	1 = display needs 16bit dummy	SYSTEM	0x2000
	Bit 14	_DISP16	1 = display is 16bit interface	SYSTEM	0x4000
	Bit 15	_PROPFONT	1 = current font is proportional	SYSTEM	0x8000

4. Appendix A : Example 4DGL Code

```
#platform "GOLDELOX-GFX2"

/*  4DGL Demo Application
    -- Scaled General Demo -
    -- Tested on uOLED-128-G1 -
    -- and uOLED160-G1 platforms --
    -- Goldelox GFX2 Platforms --
*/

#inherit "4DGL_16bitColours.fnc"

// define a custom font.
// Custom fonts can also be placed in MEDIA (ie on uSD/uSDHC card), however
// text blitting will run much faster from a data statement.
#DATA
    byte MS_SanSerif8x12
    2,              // Type 2, Char Width preceeds character; Table of widths also
    96,             // Num chars
    32,             // Starting Char
    8,              // Font_Width
    12,             // Font_Height
    4, 4, 6, 8, 7, 8, 7, 3,     // Widths of chars 0x32 to 0x39
    4, 4, 5, 7, 4, 4, 4, 6,     // etc.
    7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 4, 4, 7, 7, 7, 7,
    8, 8, 8, 8, 8, 8, 7, 8,
    8, 4, 6, 8, 7, 8, 8, 8,
    8, 8, 8, 8, 8, 8, 8, 8,
    8, 8, 8, 4, 6, 4, 7, 7,
    4, 7, 7, 7, 7, 7, 4, 7,
    7, 3, 3, 7, 3, 9, 7, 7,
    7, 7, 4, 6, 4, 7, 7, 8,
    6, 6, 6, 5, 3, 5, 8, 4,
    4, 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,   // 32 ' '
    4, 0x00,0x00,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x00,0x40,0x00,   // 33 '!'
    6, 0x00,0x00,0x48,0x48,0x48,0x00,0x00,0x00,0x00,0x00,0x00,0x00,   // 34 '"'
    8, 0x00,0x00,0x24,0x24,0x7E,0x24,0x24,0x24,0x7E,0x24,0x24,0x00,   // 35 '#'
    7, 0x00,0x00,0x10,0x38,0x54,0x50,0x30,0x18,0x14,0x54,0x38,0x10,   // 36 '$'
    8, 0x00,0x00,0x30,0x49,0x32,0x04,0x08,0x10,0x26,0x49,0x06,0x00,   // 37 '%'
    7, 0x00,0x00,0x20,0x50,0x50,0x20,0x20,0x54,0x48,0x48,0x34,0x00,   // 38 '&'
    3, 0x00,0x00,0x40,0x40,0x40,0x00,0x00,0x00,0x00,0x00,0x00,0x00,   // 39 '''
    4, 0x00,0x00,0x20,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x40,   // 40 '('
    4, 0x00,0x00,0x40,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,   // 41 ')'
    5, 0x00,0x00,0x00,0x50,0x20,0x50,0x00,0x00,0x00,0x00,0x00,0x00,   // 42 '*'
    7, 0x00,0x00,0x00,0x00,0x00,0x10,0x10,0x7C,0x10,0x10,0x00,0x00,   // 43 '+'
    4, 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x20,0x40,   // 44 ','
    4, 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x60,0x00,0x00,0x00,0x00,   // 45 '-'
    4, 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40,0x00,   // 46 '.'
    6, 0x00,0x00,0x08,0x08,0x08,0x10,0x10,0x20,0x20,0x40,0x40,0x00,   // 47 '/'
    7, 0x00,0x00,0x38,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x38,0x00,   // 48 '0'
    7, 0x00,0x00,0x10,0x70,0x10,0x10,0x10,0x10,0x10,0x10,0x10,0x00,   // 49 '1'
    7, 0x00,0x00,0x38,0x44,0x04,0x04,0x08,0x10,0x20,0x40,0x7C,0x00,   // 50 '2'
    7, 0x00,0x00,0x38,0x44,0x04,0x04,0x18,0x04,0x04,0x44,0x38,0x00,   // 51 '3'
```

```
    7, 0x00,0x00,0x08,0x18,0x18,0x28,0x28,0x48,0x7C,0x08,0x08,0x00,   // 52 '4'
    7, 0x00,0x00,0x7C,0x40,0x40,0x78,0x44,0x04,0x04,0x44,0x38,0x00,   // 53 '5'
    7, 0x00,0x00,0x38,0x44,0x40,0x40,0x78,0x44,0x44,0x44,0x38,0x00,   // 54 '6'
    7, 0x00,0x00,0x7C,0x04,0x08,0x08,0x10,0x10,0x20,0x20,0x20,0x00,   // 55 '7'
    7, 0x00,0x00,0x38,0x44,0x44,0x44,0x38,0x44,0x44,0x44,0x38,0x00,   // 56 '8'
    7, 0x00,0x00,0x38,0x44,0x44,0x44,0x3C,0x04,0x04,0x44,0x38,0x00,   // 57 '9'
    4, 0x00,0x00,0x00,0x00,0x00,0x40,0x00,0x00,0x00,0x00,0x40,0x00,   // 58 ':'
    4, 0x00,0x00,0x00,0x00,0x00,0x20,0x00,0x00,0x00,0x00,0x20,0x40,   // 59 ';'
    7, 0x00,0x00,0x00,0x00,0x08,0x10,0x20,0x40,0x20,0x10,0x08,0x00,   // 60 '<'
    7, 0x00,0x00,0x00,0x00,0x00,0x00,0x7C,0x00,0x7C,0x00,0x00,0x00,   // 61 '='
    7, 0x00,0x00,0x00,0x00,0x40,0x20,0x10,0x08,0x10,0x20,0x40,0x00,   // 62 '>'
    7, 0x00,0x00,0x38,0x44,0x04,0x04,0x08,0x10,0x10,0x00,0x10,0x00,   // 63 '?'
    8, 0x00,0x00,0x0C,0x32,0x21,0x4D,0x53,0x52,0x4C,0x20,0x31,0x0E,   // 64 '@'
    8, 0x00,0x00,0x10,0x10,0x28,0x28,0x44,0x44,0x7C,0x82,0x82,0x00,   // 65 'A'
    8, 0x00,0x00,0x78,0x44,0x44,0x44,0x78,0x44,0x44,0x44,0x78,0x00,   // 66 'B'
    8, 0x00,0x00,0x3C,0x42,0x40,0x40,0x40,0x40,0x40,0x42,0x3C,0x00,   // 67 'C'
    8, 0x00,0x00,0x78,0x44,0x42,0x42,0x42,0x42,0x42,0x44,0x78,0x00,   // 68 'D'
    8, 0x00,0x00,0x7C,0x40,0x40,0x40,0x78,0x40,0x40,0x40,0x7C,0x00,   // 69 'E'
    7, 0x00,0x00,0x7C,0x40,0x40,0x40,0x78,0x40,0x40,0x40,0x40,0x00,   // 70 'F'
    8, 0x00,0x00,0x3C,0x42,0x40,0x40,0x4E,0x42,0x42,0x46,0x3A,0x00,   // 71 'G'
    8, 0x00,0x00,0x42,0x42,0x42,0x42,0x7E,0x42,0x42,0x42,0x42,0x00,   // 72 'H'
    4, 0x00,0x00,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x00,   // 73 'I'
    6, 0x00,0x00,0x10,0x10,0x10,0x10,0x10,0x10,0x90,0x90,0x60,0x00,   // 74 'J'
    8, 0x00,0x00,0x44,0x48,0x50,0x60,0x60,0x50,0x48,0x44,0x42,0x00,   // 75 'K'
    7, 0x00,0x00,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x7C,0x00,   // 76 'L'
    8, 0x00,0x00,0x41,0x41,0x63,0x63,0x55,0x55,0x49,0x49,0x41,0x00,   // 77 'M'
    8, 0x00,0x00,0x42,0x62,0x62,0x52,0x52,0x4A,0x46,0x46,0x42,0x00,   // 78 'N'
    8, 0x00,0x00,0x3C,0x42,0x42,0x42,0x42,0x42,0x42,0x42,0x3C,0x00,   // 79 'O'
    8, 0x00,0x00,0x7C,0x42,0x42,0x42,0x7C,0x40,0x40,0x40,0x40,0x00,   // 80 'P'
    8, 0x00,0x00,0x3C,0x42,0x42,0x42,0x42,0x42,0x4A,0x46,0x3C,0x02,   // 81 'Q'
    8, 0x00,0x00,0x7C,0x42,0x42,0x42,0x7C,0x42,0x42,0x42,0x42,0x00,   // 82 'R'
    8, 0x00,0x00,0x38,0x44,0x40,0x40,0x38,0x04,0x04,0x44,0x38,0x00,   // 83 'S'
    8, 0x00,0x00,0x7C,0x10,0x10,0x10,0x10,0x10,0x10,0x10,0x10,0x00,   // 84 'T'
    8, 0x00,0x00,0x42,0x42,0x42,0x42,0x42,0x42,0x42,0x42,0x3C,0x00,   // 85 'U'
    8, 0x00,0x00,0x41,0x41,0x22,0x22,0x22,0x14,0x14,0x08,0x08,0x00,   // 86 'V'
    8, 0x00,0x00,0x41,0x41,0x41,0x22,0x2A,0x2A,0x1C,0x14,0x14,0x00,   // 87 'W'
    8, 0x00,0x00,0x41,0x41,0x22,0x14,0x08,0x14,0x22,0x41,0x41,0x00,   // 88 'X'
    8, 0x00,0x00,0x41,0x41,0x22,0x14,0x08,0x08,0x08,0x08,0x08,0x00,   // 89 'Y'
    8, 0x00,0x00,0x7F,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x7F,0x00,   // 90 'Z'
    4, 0x00,0x00,0x60,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x40,   // 91 '['
    6, 0x00,0x00,0x40,0x40,0x40,0x20,0x20,0x10,0x10,0x08,0x08,0x00,   // 92 '\'
    4, 0x00,0x00,0x60,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,   // 93 ']'
    7, 0x00,0x10,0x28,0x44,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,   // 94 '^'
    7, 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,   // 95 '_'
    4, 0x00,0x00,0x40,0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,   // 96 '`'
    7, 0x00,0x00,0x00,0x00,0x00,0x38,0x04,0x3C,0x44,0x44,0x3C,0x00,   // 97 'a'
    7, 0x00,0x00,0x40,0x40,0x40,0x78,0x44,0x44,0x44,0x44,0x78,0x00,   // 98 'b'
    7, 0x00,0x00,0x00,0x00,0x00,0x38,0x44,0x40,0x40,0x44,0x38,0x00,   // 99 'c'
    7, 0x00,0x00,0x04,0x04,0x04,0x3C,0x44,0x44,0x44,0x44,0x3C,0x00,   // 100 'd'
    7, 0x00,0x00,0x00,0x00,0x00,0x38,0x44,0x7C,0x40,0x44,0x38,0x00,   // 101 'e'
    4, 0x00,0x00,0x20,0x40,0x40,0x60,0x40,0x40,0x40,0x40,0x40,0x00,   // 102 'f'
    7, 0x00,0x00,0x00,0x00,0x00,0x3C,0x44,0x44,0x44,0x44,0x3C,0x04,   // 103 'g'
    7, 0x00,0x00,0x40,0x40,0x40,0x58,0x64,0x44,0x44,0x44,0x44,0x00,   // 104 'h'
    3, 0x00,0x00,0x40,0x00,0x00,0x40,0x40,0x40,0x40,0x40,0x40,0x00,   // 105 'i'
    3, 0x00,0x00,0x40,0x00,0x00,0x40,0x40,0x40,0x40,0x40,0x40,0x40,   // 106 'j'
    7, 0x00,0x00,0x40,0x40,0x40,0x48,0x50,0x60,0x50,0x48,0x44,0x00,   // 107 'k'
    3, 0x00,0x00,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x00,   // 108 'l'
    9, 0x00,0x00,0x00,0x00,0x00,0x76,0x49,0x49,0x49,0x49,0x49,0x00,   // 109 'm'
```

```
    7, 0x00,0x00,0x00,0x00,0x00,0x58,0x64,0x44,0x44,0x44,0x44,0x00,   // 110 'n'
    7, 0x00,0x00,0x00,0x00,0x00,0x38,0x44,0x44,0x44,0x44,0x38,0x00,   // 111 'o'
    7, 0x00,0x00,0x00,0x00,0x00,0x78,0x44,0x44,0x44,0x44,0x78,0x40,   // 112 'p'
    7, 0x00,0x00,0x00,0x00,0x00,0x3C,0x44,0x44,0x44,0x44,0x3C,0x04,   // 113 'q'
    4, 0x00,0x00,0x00,0x00,0x00,0x60,0x40,0x40,0x40,0x40,0x40,0x00,   // 114 'r'
    6, 0x00,0x00,0x00,0x00,0x00,0x30,0x48,0x20,0x10,0x48,0x30,0x00,   // 115 's'
    4, 0x00,0x00,0x00,0x40,0x40,0x60,0x40,0x40,0x40,0x40,0x20,0x00,   // 116 't'
    7, 0x00,0x00,0x00,0x00,0x00,0x44,0x44,0x44,0x44,0x4C,0x34,0x00,   // 117 'u'
    7, 0x00,0x00,0x00,0x00,0x00,0x44,0x44,0x28,0x28,0x10,0x10,0x00,   // 118 'v'
    8, 0x00,0x00,0x00,0x00,0x00,0x49,0x49,0x55,0x55,0x22,0x22,0x00,   // 119 'w'
    6, 0x00,0x00,0x00,0x00,0x00,0x48,0x48,0x30,0x30,0x48,0x48,0x00,   // 120 'x'
    6, 0x00,0x00,0x00,0x00,0x00,0x48,0x48,0x48,0x48,0x30,0x20,0x20,   // 121 'y'
    6, 0x00,0x00,0x00,0x00,0x00,0x78,0x08,0x10,0x20,0x40,0x78,0x00,   // 122 'z'
    5, 0x00,0x10,0x20,0x20,0x20,0x20,0x40,0x20,0x20,0x20,0x20,0x10,   // 123 '{'
    3, 0x00,0x00,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x40,   // 124 '|'
    5, 0x00,0x40,0x20,0x20,0x20,0x20,0x10,0x20,0x20,0x20,0x20,0x40,   // 125 '}'
    8, 0x00,0x00,0x00,0x32,0x4C,0x00,0x00,0x00,0x00,0x00,0x00,0x00,   // 126 '~'
    4, 0x00,0x00,0x00,0x60,0x60,0x60,0x60,0x60,0x60,0x60,0x60,0x60    // 127 '□'
#END

// a message for the moving banner
#DATA
    byte message "......Goldelox GFX2 Graphics......",0
#END

// the 'wall' colours
#CONST
    LEFTCOLOUR      0xF800
    RIGHTCOLOUR     0xFFFF
    TOPCOLOUR       0x001F
    BOTTOMCOLOUR    0x07E0
#END

// constants for the view-port
// These may need adjusting for smaller displays
#CONST
    windowXpos      30
    windowYpos      30
    windowWidth     110
    windowHeight    60
#END

// object types.
// 2,3,4,5 and 6 doubles as polygon vertices counts
#CONST
    RANDOM          0
    CIRCLE          1
    LINE            2
    TRIANGLE        3
    RECTANGLE       4
    PENTAGON        5
    HEXAGON         6
#END



// 'ball' speed factors determine
// how many pixels to jump per movement
```

```
#constant XSPEED        3
#constant YSPEED        2

// the width of the side walls
#constant WALLWIDTH     2

// 'ball' object radius
#constant BALLSIZE 4

// global working variables
var ball_x, ball_y, ball_r, ball_colour;
var xdir, ydir, xspeed, yspeed;
var screenwidth, screenheight, xc, yc;
var tophit, bottomhit, lefthit, righthit;
var windowLeft, windowTop, windowRight, windowBottom;
var angle, newseed;

// global variables for the polygon generator
var Xcoords[6], Ycoords[6];     // big enough for a hexagon
var targetX, targetY;           // targets for orbit

// array of pointers for text messages
var messages[4];


// polyline array for scope
#constant SAMPLES 20
var ScopeBufX[SAMPLES];
var ScopeBufY[SAMPLES];
var freq[4];

var mediaflag;      // set to 1 if uSD/uSDHC card detected

//=====================================================================================
// draw random waveform
//=====================================================================================
func doRandScope(var samples, var colr, var smpl)
    var w,h,n,xstep,yoffs,x,yscale,xoffs,seedoffs;
    w := windowRight-windowLeft;
    h := windowBottom-windowTop;
    xstep:=w/samples+1;
    yscale:=h/2;
    yoffs:=h/2+windowTop;
    SEED(smpl);
    x:=windowLeft;
    while (n<samples)
    // undraw the old sample as we create new one (looks better, less flicker)
        gfx_Line(ScopeBufX[n],ScopeBufY[n],ScopeBufX[n+1],ScopeBufY[n+1],BLACK);
        ScopeBufY[n]:=(RAND()%yscale)+yoffs;
        ScopeBufX[n]:=x;
        x := x+xstep;
        n++;
    wend
    gfx_Polyline(samples, ScopeBufX, ScopeBufY, colr);  // draw the new sample
endfunc
//=====================================================================================
// draw a sinewave
//=====================================================================================
```

```
func doSineScope(var samples, var colr, var smpl)
    var w,h,n,xstep,yoffs,x,yscale,xoffs,seedoffs;
    w := windowRight-windowLeft;
    h := windowBottom-windowTop;
    xstep:=w/samples+1;
    yscale:=h/2;
    yoffs:=h/2+windowTop;
    x:=windowLeft;
    gfx_Polyline(samples, ScopeBufX, ScopeBufY, BLACK);  // undraw the old
                                                         // buffer first
    while (n<samples)
        ScopeBufY[n]:=SIN(xoffs)/4+yoffs;
        ScopeBufX[n]:=x;
        x := x+xstep;
        xoffs := xoffs+smpl;
        n++;
    wend
    gfx_Polyline(samples, ScopeBufX, ScopeBufY, colr);;  // draw the new sample
endfunc


//=============================================================================
// build a polygon with a number of sides determined by var "sides"
// around the current origin. The distance from the origin to the
// equidistent vertices from origin determined
// by var "distance". var "angle" is the starting angle for the
// first virtices. Draws the polygon in colour var "colr"
// NB make sure the array is big enough for the required number of sides
//=============================================================================
func MakePolygon(var angle, var sides, var distance, var colr)
    var index, step;
    index := 0;
    step := 360/sides;                  // work out the step size
    while (sides < 360)                 // until we do a complete polygon
        gfx_Orbit(angle, distance);
        Xcoords[index] := targetX;      // build a polygon in the matrix
        Ycoords[index] := targetY;
        index++;
        angle := angle + step;
        sides := sides + step;
    wend
    gfx_Polygon(index, Xcoords, Ycoords, colr);
endfunc


//=============================================================================
// ball object control
//=============================================================================
func DrawBall(var type, var colour)
    var count;

    gosub(type),(
                circle,
                text,
                triangle,
                rectangle,
                pentagon,
                hexagon,
```

```
                random
                );
                goto default;  // unknown type default exit

// case circle
circle:
    gfx_CircleFilled(ball_x, ball_y, BALLSIZE, colour);   // redraw the ball
    endsub;

// case text
text:
    txt_Opacity(TRANSPARENT);   // transparent text
    txt_FontID(0);              // default small font
    //txt_FGcolour(RAND());
    txt_FGcolour(colour);
    gfx_MoveTo(ball_x, ball_y);    // draw a pixel trail
    putstr("4DGL");
    endsub;


// these cases same, type is used to determine number of sides
triangle:
rectangle:
pentagon:
hexagon:
    gfx_MoveTo(ball_x, ball_y);             // using the balls origin
    MakePolygon(angle, type, 10, colour);   // make 3 sided polygon = triangle
    endsub;

// case random
random:
    if (colour)
        SEED(newseed);
        gfx_ObjectColour(RAND()|0x8408);    // ensure hi colours
    else
        SEED(newseed++);
        RAND();            // RAND here to compensate so we get  repeat sequence
        gfx_ObjectColour(BLACK);
    endif

    count := 5;
    while (count--)
        gfx_MoveTo(ball_x+RAND()%15, ball_y+RAND()%15);
        //gfx_Dot();                                        // draw a pixel trail
        gfx_Bullet(3);                                      // draw random circles
        //gfx_BoxTo(ball_x, ball_y);                        // draw random boxes
    wend
    endsub;

default:

endfunc

//=============================================================================
// part of intro, fill clipped area with pixels then remove in same orded
//=============================================================================
func doDots()
    var n,x,y,w,h;
```

```
    // random dots
    SEED(1234);
    w := windowRight - windowLeft;
    h := windowBottom - windowTop;
    n := -3000;
    while (n++<3000)
        x := ABS(RAND()%w) + windowLeft+1;
        y := ABS(RAND()%h) + windowTop+1;
        gfx_PutPixel(x , y , RAND());
    wend

    // undraw the dots
    SEED(1234);
    n := -3000;
    while (n++<3000)
        x := ABS(RAND()%w) + windowLeft+1;
        y := ABS(RAND()%h) + windowTop+1;
        RAND();
        gfx_PutPixel(x , y , 0);
    wend
endfunc

//=============================================================================
// part of intro, fill entire screen with lines then remove in same orded
// Note that clipping will take care of line endpoints outside to clipping area
//=============================================================================
func doLines()
    var n;
    // random lines
    SEED(9876);
    n := -200;
    while (n++<200)
        gfx_Line(ABS(RAND()%screenwidth), ABS(RAND()%screenheight), ABS(RAND()
        %screenwidth), ABS(RAND()%screenheight), RAND());
    wend

    // undraw the lines
    SEED(9876);
    n := -200;
    while (n++<200)
        gfx_Line(ABS(RAND()%screenwidth), ABS(RAND()%screenheight), ABS(RAND()
        %screenwidth), ABS(RAND()%screenheight), 0);
        RAND();
    wend
endfunc

//=============================================================================
// Check the baal position against the walls.
// Change direction registers accordingly.
//=============================================================================
func collision()
    if(ball_x <= lefthit)
       ball_x := lefthit;
       ball_colour := LEFTCOLOUR;
       xdir := -xdir;
    endif

    if(ball_x >= righthit)
```

```
       ball_x := righthit;
       ball_colour := RIGHTCOLOUR;
       xdir := -xdir;
    endif

    if (ball_y <= tophit)
       ball_y := tophit;
       ball_colour := TOPCOLOUR;
       ydir := -ydir;
    endif

    if(ball_y >= bottomhit)
       ball_y := bottomhit;
       ball_colour := BOTTOMCOLOUR;
       ydir := -ydir;
    endif
endfunc

//=============================================================================
// EVE starts executing code from here
//=============================================================================
func main()
    var mode, timer, obj, scrollpos, n, linepattern, intro, intronum, scopeloop;

    if (media_Init() == 0)               // initialise and test the uSD/uSDHC
card
        print("No uSD CARD Installed\n");
        print("Some demo's are disabled");
        pause(2000);
        gfx_Cls();
    endif

    mode := 0;
    linepattern := 0xF0F0;
    messages[0] := " LANDSCAPE";
    messages[1] := "LANDSCAPE_R";
    messages[2] := " PORTRAIT";
    messages[3] := "PORTRAIT_R";

    //gfx_Set(CONTRAST, 16);
    gfx_Contrast(16);   // max. brightness
    gfx_Cls();

    // set generic target variables for the orbit command
    gfx_OrbitInit(&targetX, &targetY);
    txt_Set(FONT_ID, MS_SanSerif8x12);   // don't use default system font, use
                                         // font provided
repeat
    timer := 0;                         // timer for SCREEN_MODE switching
    gfx_Cls();
    gfx_Set(SCREEN_MODE, mode);     // set required screen mode

    // this is mainly for 'non square' display to make the ball speed realistic
    if (mode < 2)
        xspeed := XSPEED;           // keep correct ball speed aspect
        yspeed := YSPEED;
    else
        xspeed := YSPEED;
```

```
        yspeed := XSPEED;
    endif

    // get the display parameters
    screenwidth := peekB(GFX_XMAX);
    screenheight := peekB(GFX_YMAX);

    // determine the centre point
    xc := screenwidth >> 1;
    yc := screenheight >> 1;
    ball_colour := WHITE;                          // initial ball colour
    xdir := 1; ydir := 1;                          // initial ball direction
    ball_x := 20; ball_y := 20;                    // initial ball position

    // draw the walls
    // draw Top Wall
    gfx_RectangleFilled(0, 0, screenwidth-1, WALLWIDTH-1, TOPCOLOUR);

    // Draw Bottom Wall
    gfx_RectangleFilled(0, screenheight-WALLWIDTH, screenwidth-1, screenheight-
    1, BOTTOMCOLOUR);

    // Draw Left Wall
    gfx_RectangleFilled(0, WALLWIDTH-1, WALLWIDTH-1, screenheight-WALLWIDTH-1,
    LEFTCOLOUR);

    // Draw Right Wall
    gfx_RectangleFilled(screenwidth-WALLWIDTH, WALLWIDTH, screenwidth-1,
    screenheight-WALLWIDTH-1, RIGHTCOLOUR);

    // calculate the collision positions
    tophit := WALLWIDTH+BALLSIZE;
    bottomhit := screenheight-WALLWIDTH-BALLSIZE-1;
    lefthit := WALLWIDTH+BALLSIZE;
    righthit := screenwidth-WALLWIDTH-BALLSIZE-1;

    // set clipping area
    windowLeft := lefthit;
    windowTop := tophit+10;
    windowRight := righthit - 16;
    windowBottom := bottomhit -40;

    // preset the clipping area, activated later...
    gfx_ClipWindow(windowLeft, windowTop, windowRight, windowBottom);

    // draw a rectangle around the clipped area
    gfx_Rectangle(windowLeft-1, windowTop-1, windowRight+1, windowBottom+1,
    YELLOW);
    //  test: draw a small outline rectangle outside
    gfx_Rectangle(windowLeft+5, windowBottom+10, windowLeft+15, windowBottom+20,
    RED);
    // test: draw a small solid rectangle outside
    gfx_RectangleFilled(windowLeft+20, windowBottom+10, windowLeft+30,
    windowBottom+20, GREEN);

    // test: draw a small outline circle
    gfx_Circle(windowLeft+40, windowBottom+15, 5, BLUE);
```

```
    // test: draw a small filled circle
    gfx_CircleFilled(windowLeft+60, windowBottom+15, 5, YELLOW);

    gfx_Set(CLIPPING, OFF);      // turn off clipping so we can print outside
                                 // the clip region
    txt_FGcolour(RED);
    txt_BGcolour(YELLOW);
    txt_Bold(ON);
    //txt_FontID(2);
    //txt_Set(TEXT_ITALIC, ON);
    //txt_Set(TEXT_OPACITY, TRANSPARENT);    // transparent text is faster
    //gfx_MoveTo(xc-50, yc+20);
    gfx_MoveTo(xc-50, bottomhit -12);
    print(mode," ",[STR] messages[mode]);
    gfx_Set(CLIPPING, ON);                   // turn on clipping

    // decide which intro we use for the next screen
    if (intro)
      intro := 0;
      // clear the clipped area
      gfx_RectangleFilled(windowLeft,windowTop,windowRight,windowBottom, BLACK);
      intronum++;
      if (intronum == 1)
          n:=-180;
          while(n<180)
              doSineScope(SAMPLES, YELLOW, n++);
              n++;
              //pause(10);
          wend
      else if (intronum == 2)
          n:=200;
          while(n)
              doRandScope(SAMPLES, BLUE, n--);
              //pause(10);
          wend
      else if (intronum == 3)
          doLines();
      else
          doDots();
          intronum := 0;
      endif
      gfx_RectangleFilled(windowLeft,windowTop,windowRight,windowBottom,BLACK);
    endif

   // timer0 is the screen mode change timer
   *TIMER0 := 7000;
   repeat
      // draw a cross through the clipped area box
      gfx_LinePattern(linepattern);
      gfx_Line(windowLeft+1,windowTop+1,windowRight-1,windowBottom-1, MAGENTA);
      gfx_Line(windowLeft+1,windowBottom-1,windowRight-1,windowTop+1, MAGENTA);
      gfx_LinePattern(0);

      // timer2 is used for the banner scrolling
      if (!*TIMER2)
         *TIMER2 := 50;
         txt_Opacity(OPAQUE);            // transparent text
         txt_FontID(0);                  // default system font
```

```
         gfx_Clipping(OFF);
         gfx_ClipWindow(windowLeft+10,WALLWIDTH,windowRight-10,WALLWIDTH+8);
         gfx_Clipping(ON);
         scrollpos := scrollpos-1;
         n:=strwidth(message);
         if(scrollpos < windowLeft+10-n) scrollpos := windowRight-10;
         gfx_MoveTo(scrollpos, WALLWIDTH+2);
         txt_FGcolour(WHITE);
         txt_BGcolour(DARKGREEN);
         //txt_Italic(ON);
         txt_Bold(ON);
         putstr(message);
         gfx_Clipping(OFF);
         gfx_ClipWindow(windowLeft, windowTop+1, windowRight, windowBottom);
         gfx_Clipping(ON);
      endif

      // timer3 is used to shift the line pattern
      if(!*TIMER3)
         *TIMER3 := 100;
         linepattern := linepattern << 1;
         if (OVF())  linepattern := linepattern | 1;
      endif

      // timer 0 is for ball timing
      if(!*TIMER1)
         *TIMER1 := 30;
         DrawBall(obj, BLACK);                  // erase the ball object
         angle := angle + 10;

         ball_x := ball_x + xdir * xspeed;
         ball_y := ball_y + ydir * yspeed;

         collision();                           // detect collision
         DrawBall(obj, ball_colour);            // redraw the ball object
         //DrawBall(obj, RAND());               // redraw the ball object
      endif
   until (!*TIMER0)

   scrollpos := windowLeft+10; // reset the banner
   if (++mode > 3)
      mode := 0;                                // next screen mode
      if (obj++ > HEXAGON) obj:=0;             // nextball object
      intro := 1;                               // set flag so we do the intro
   endif

 forever  // start again
endfunc
//=============================================================================
```

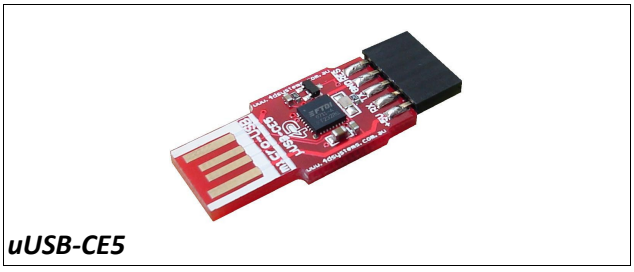
5. Appendix B : Development and Support Tools

5.1 PmmC Loader – PmmC File Programming Software Tool

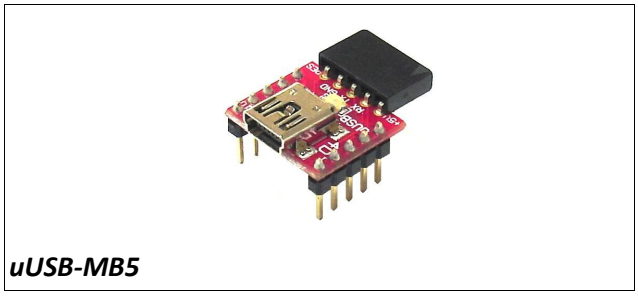
The 'PmmC Loader' is a free software tool for Windows based PC platforms. Use this tool to program the latest PmmC file into the GOLDELOX-GFX2 chip embedded in your application board. It is available for download from the 4D Systems website, www.4dsystems.com.au

5.2 microUSB – PmmC Programming Hardware Tool

The micro-USB module is a USB to Serial bridge adaptor that provides a convenient physical link between the PC and the GOLDELOX-GFX2 device. A range of custom made micro-USB devices such as the uUSB-MB5 and the uUSB-CE5 are available from 4D Systems www.4dsystems.com.au. The micro-USB module is an essential hardware tool for all the relevant software support tools to program, customise and test the GOLDELOX-GFX2 chip.



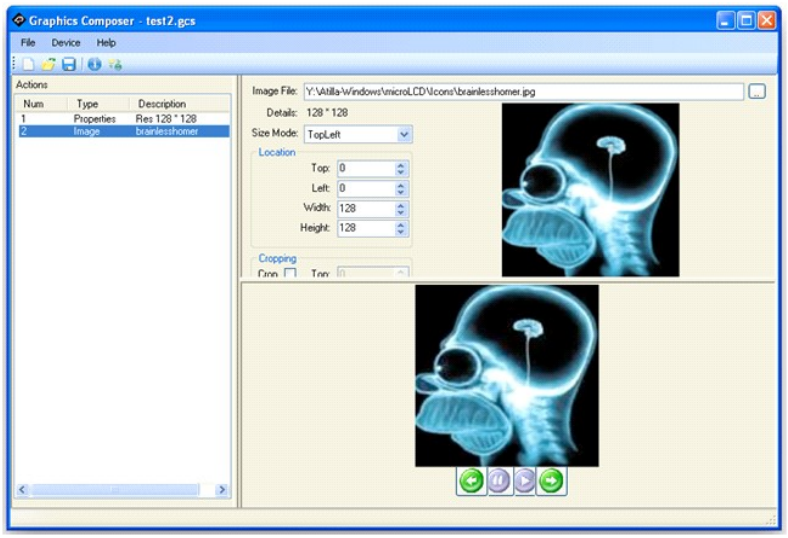
uUSB-CE5



uUSB-MB5

5.3 Graphics Composer – Software Tool

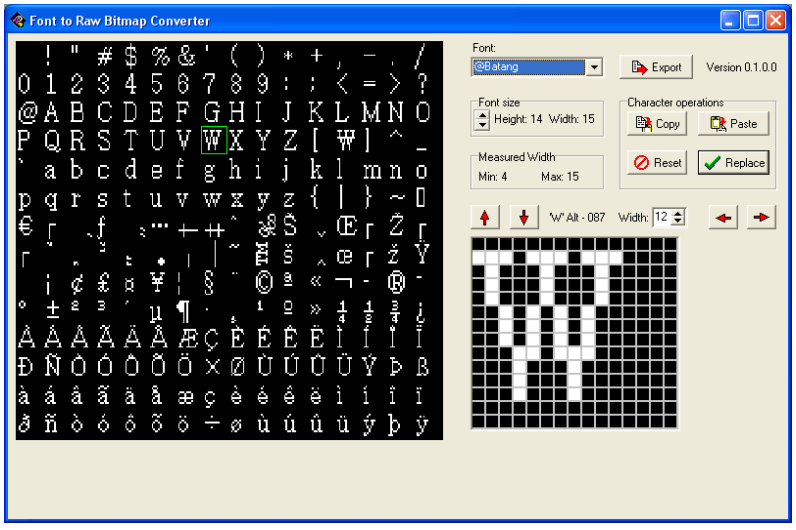
The Graphics Composer is a free software tool for Windows. This software tool is an aid to composing a slide show of images/animations/movie-clips (multi-media objects) which can then be downloaded into the SD/uSD/uSDHC/MMC memory card that is supported by the GOLDELOX-GFX2. The multimedia objects can then be called within the user application 4DGL program. It is available for download from the 4D Systems website, www.4dsystems.com.au



5.4 FONT Tool – Software Tool

Font-Tool is a free software tool for Windows based PC platforms. Use this tool to assist in the conversion of standard Windows fonts (including True Type) into the bitmap fonts used by the GOLDELOX-GFX2 chip. It is available for download from the 4D Systems website, www.4dsystems.com.au.

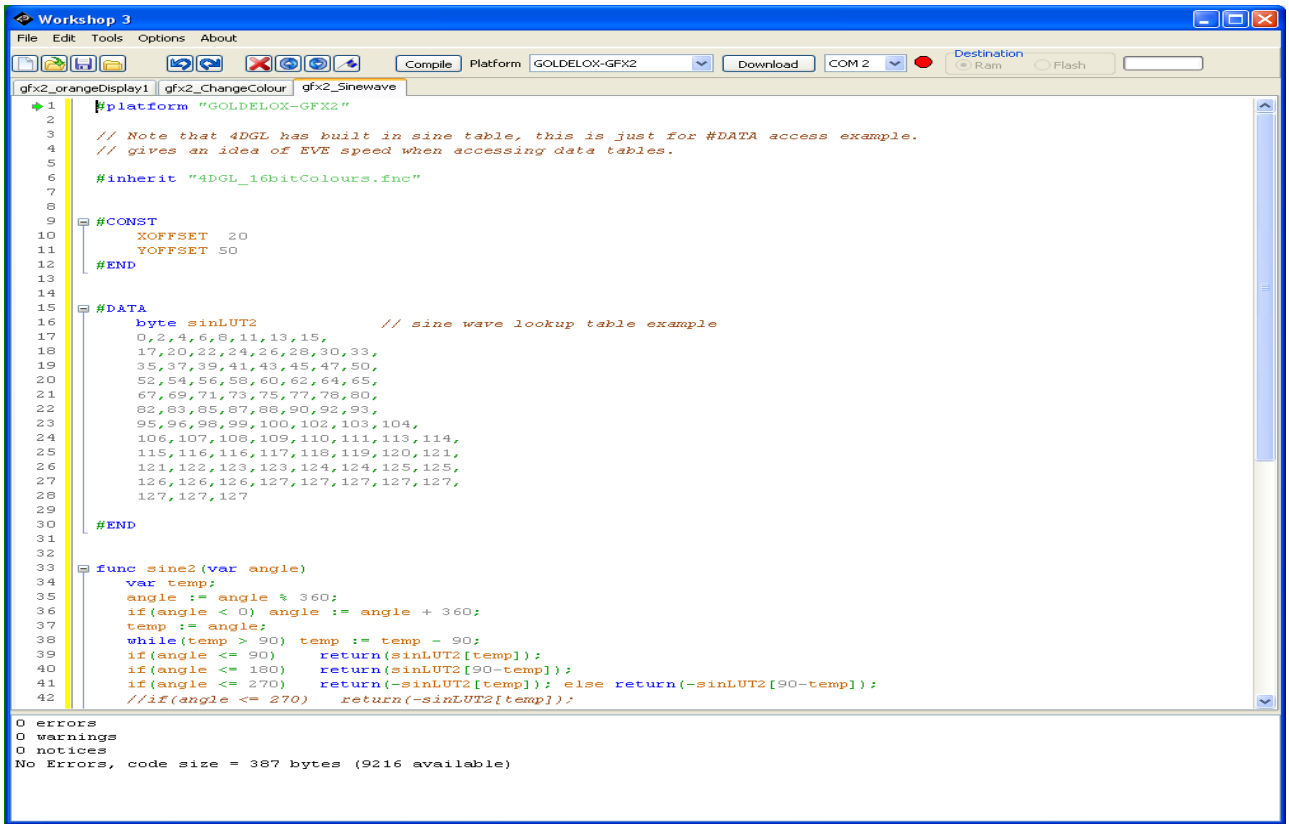
Disclaimer: Windows fonts may be protected by copyright laws. This software is provided for experimental purposes only.



5.5 4DGL-Workshop3–Complete IDE Editor, Compiler, Linker, DownLoader

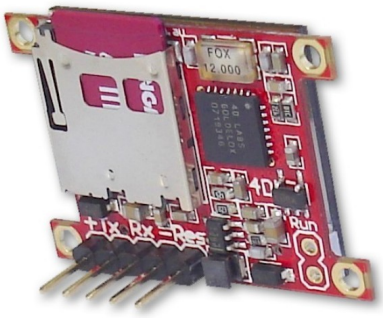
The 4DGL-Workshop3 IDE provides an integrated software development environment for all of the 4D family of processors and modules. The IDE combines the Editor, Compiler, Linker and DownLoader to develop complete 4DGL application code. All user application code is developed within the Workshop IDE.

It is available for download from the 4D Systems website, www.4dsystems.com.au

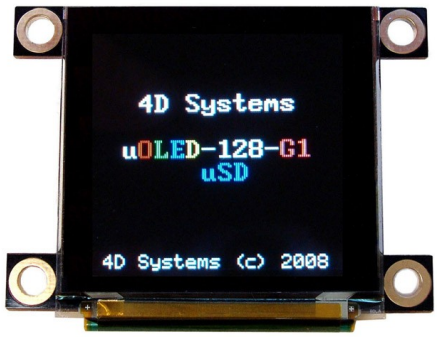


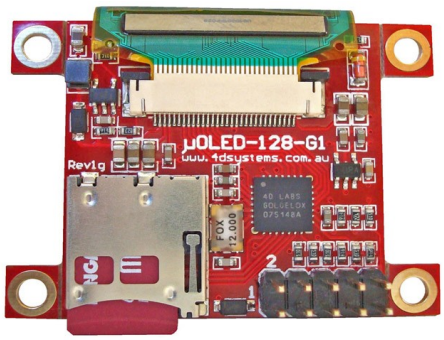
5.6 Evaluation Display Modules

The following modules, available from 4D Systems, are ideal evaluation platforms to discover what the GOLDELOX-GFX2 processor has to offer.

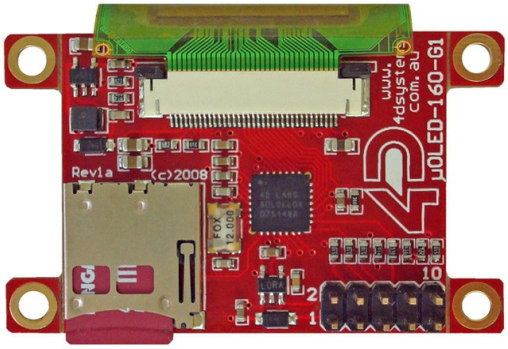


uOLED-96-G1(GFX) 0.96" PMOLED, 65K Colour, 4DGL Platform Display Module





uOLED-128-G1(GFX) 1.5" PMOLED, 65K Colour, 4DGL Platform Display Module



uOLED-160-G1(GFX) 1.7" PMOLED, 65K Colour, 4DGL Platform Display Module

Proprietary Information

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

4D Labs endeavours to ensure that the information in this document is correct and fairly stated but does not accept liability for any error or omission. The development of 4D Labs products and services is continuous and published information may not be up to date. It is important to check the current position with 4D Labs.

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

Disclaimer of Warranties & Limitation of Liability

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

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

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

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