4D LABS SEMICONDUCTORS

PICASO

Embedded Graphics Processor

INTERNAL FUNCTIONS

Document Revision: 7.1

Document Date: 21st March 2019

PICASO Graphics Processor – Internal Functions

Table of Contents

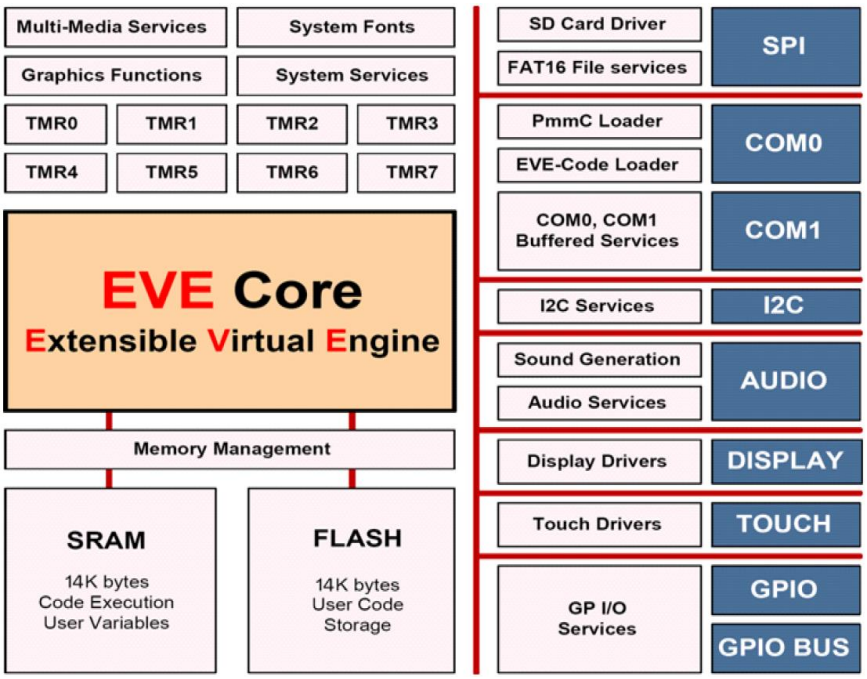
1. 4DGL Introduction

The 4D-Labs family of embedded graphics processors (Goldelox, Picaso, Diablo16, PIXXI-28 and PIXXI-44) 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 Picaso Processor. This document should be used in conjunction with ***"4DGL-Programmers-Reference-Manual"*** document.



Picaso Internal Block Diagram

2. Picaso Chip-Resident Functions Summary

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

2.1 GPIO Functions:

- pin_Set(mode, pin)
  - OUTPUT, INPUT
- pin_HI(pin)
- pin_LO(pin)
- pin_Read(pin)
- bus_In()
- bus_Out("var")
- bus_Set("var")
- bus_Write("var")
- bus_Read("var")

2.2 System Memory Access Functions:

- peekW(address)
- pokeW(address, word_value)

2.3 Maths Functions:

- ABS(value)
- MIN(value1, value2)
- MAX(value1, value2)
- SWAP(&var1, &var2)
- SIN(angle)
- COS(angle)
- RAND()
- SEED(number)
- SQRT(number)
- OVF ()
- CY()
- umul_1616(&res32, val1, val2)
- uadd_3232(&res32, &val1, &val2)
- usub_3232(&res32, &val1, &val2)
- ucmp_3232(&val1, &val2)

2.4 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) [deprecated]
- txt_Opacity(mode)
- txt_Bold(mode)
- txt_Italic(mode)
- txt_Inverse(mode)
- txt_Underlined(mode)
- txt_Attributes(value)
- txt_Wrap(value)

2.5 CType Functions:

- isdigit(char)
- isxdigit(char)
- isupper(char)
- islower(char)
- isalpha(char)
- isalnum(char)
- isprint(char)
- isspace(char)
- iswhite(char)
- toupper(char)
- tolower(char)
- LObyte(var)
- HIbyte(var)
- ByteSwap(var)

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_Ellipse(x, y, xrad, yrad, colour)
- gfx_EllipseFilled(x, y, xrad, yrad, colour)
- gfx_Button(state, x, y, buttonColour, textColour, font, textWidth, textHeight, text)
- gfx_Panel(state, x, y, width, height, colour)
- gfx_Slider(mode, x1, y1, x2, y2, colour, scale, value)
- gfx_ScreenCopyPaste(xs, ys, xd, yd, width, height)
- gfx_RGBto565(RED, GREEN, BLUE)
- gfx_332to565(COLOUR8BIT)
- gfx_TriangleFilled(x1, y1, x2, y2, x3, y3, colr)
- gfx_PolygonFilled(n, &vx, &vy, colr)
- gfx_Origin(x, y)
- gfx_Get(mode)
- gfx_ClipWindow(x1, y1, x2, y2)
- gfx_Set(function, value)

  gfx_Set shortcuts:
  - gfx_PenSize(mode)
  - gfx_BGcolour(colour)
  - gfx_ObjectColour(colour)
  - gfx_Clipping(mode)
  - gfx_TransparentColour(colour)
  - gfx_Transparency(mode)
  - gfx_FrameDelay(delay)
  - gfx_ScreenMode(delay)
  - gfx_OutlineColour(colour)
  - gfx_Contrast(value)
  - gfx_LinePattern(pattern)
  - gfx_ColourMode(mode)
  - gfx_BevelWidth(mode)
  - gfx_BevelShadow(value)
  - gfx_Xorigin(offset)
  - gfx_Yorigin(offset)

2.7 Display I/O Functions:

- disp_SetReg(register, data)
- disp_setGRAM(x1, y1, x2, y2)
- disp_WrGRAM(colour)
- disp_WriteControl(value)
- disp_WriteWord(value)
- disp_ReadWord()
- disp_Sync(line)
- disp_Disconnect()
- disp_Init()

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

- media_Init()
- media_SetAdd(HIword, LOword)
- media_SetSector(HIword, LOword)
- media_RdSector(Destination_Address)
- media_WrSector(Source_Address)
- 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:

- setbaud(rate)
- com_SetBaud(comport, baudrate/10)
- serin() or serin1()
- serout(char) or serout1(char)
- com_Init(buffer, buffsize, qualifier) or com1_Init(buffer, buffsize, qualifier)
- com_Reset() or com1_Reset()
- com_Count() or com1_Count()
- com_Full() or com1_Full()
- com_Error() or com1_Error()
- com_Sync() or com1_Sync()
- com_TXbuffer(buf, bufsize,pin) or com1_TXbuffer(buf, bufsize,pin)
- com_TXcount() or com1_TXcount()
- com_TXemptyEvent(function) or com1_TXemptyEvent(function)

2.12 I2C BUS Master Function

- func I2C_Open(Speed)
- func I2C_Close()
- func I2C_Start()
- func I2C_Stop()
- func I2C_Restart()
- func I2C_Read()
- func I2C_Write(byte)
- func I2C_Ack()
- func I2C_Nack()
- func I2C_AckStatus()
- func I2C_AckPoll(control)
- func I2C_Idle()
- func I2C_Gets(buffer, size)
- func I2C_Getn(buffer, size)
- func I2C_Puts(buffer)
- func I2C_Putn(buffer,count)

2.13 Timer Functions:

- sys_T()
- sys_T_HI()
- sys_SetTimer(timernum, value)

- sys_GetTimer(timernum)
- sys_SetTimerEvent("timernum","function")
- sys_EventQueue()
- sys_EventsPostpone()
- sys_EventsResume()
- sys_DeepSleep(units)
- sys_Sleep(units)
- iterator(offset)

2.14 FAT16 File Functions:

- file_Error()
- file_Count(filename)
- file_Dir(filename)
- file_FindFirst(fname)
- file_FindNext()
- file_Exists(fname)
- file_Open(fname, mode)
- file_Close(handle)
- file_Read(destination, size, handle)
- file_Seek(handle, HiWord, LoWord)
- file_Index(handle, Hisize, Losize, recordnum)
- file_Tell(handle, &HiWord, &LoWord)
- file_Write(Source, size, handle)
- file_Size(handle, &HiWord, &LoWord)
- file_Image(x, y, handle)
- file_ScreenCapture(x, y, width, height, handle)
- file_PutC(char, handle)
- file_GetC(handle)
- file_PutW(word, handle)
- file_GetW(handle)
- file_PutS(source, handle)
- file_GetS(*String, size, handle)
- file_Erase(fname)
- file_Rewind(handle)
- file_LoadFunction(fname.4XE)
- file_Run(fname..4XE, arglistptr)
- file_Exec(fname..4XE, arglistptr)
- file_LoadImageControl(fname1, fname2, mode)
- file_Mount()
- file_Unmount()
- file_PlayWAV

2.15 Sound Control Functions:

- Snd_Volume(var)
- Snd_Pitch(pitch)
- Snd_BufSize(var)
- Snd_Stop()
- Snd_Pause()
- Snd_Continue()
- Snd_Playing()

2.16 String Class Functions:

- str_Ptr(&var)
- str_GetD(&ptr, &var)

- str_GetW(&ptr, &var)
- str_GetHexW(&ptr, &var)
- str_GetC(&ptr, &var)
- str_GetByte(ptr)
- str_GetWord(ptr)
- str_PutByte(ptr, val)
- str_PutWord(ptr, val)
- str_Match(&ptr, *str)
- str_MatchI(&ptr, *str)
- str_Find(&ptr, *str)
- str_FindI(&ptr, *str)
- str_Length(ptr)
- str_Printf(&ptr, *format)
- str_Cat(&destination, &Source)
- str_CatN(&ptr, str, count)
- str_ByteMove(src, dest, count)
- str_Copy(dest, src)
- str_CopyN(dest, src, count)

2.17 Touch Screen Functions: (Touch functions do not apply to uVGA-II/III modules)

- touch_DetectRegion(x1, y1, x2, y2)
- touch_Set(mode)
- touch_Get(mode)

2.18 Image Control Functions:

- img_SetPosition(handle, index, xpos, ypos)
- img_Enable(handle, index)
- img_Disable(handle, index)
- img_Darken(handle, index)
- img_Lighten(handle, index)
- img_SetWord(handle, index, offset, word)
- img_GetWord(handle, index, offset)
- img_Show(handle, index)
- img_SetAttributes(handle, index, value)
- img_ClearAttributes(handle, index, value)
- img_Touched(handle, index)

2.19 Memory Allocation Functions:

- mem_Alloc(size)
- mem_Allocv(size)
- mem_Allocz(size)
- mem_Realloc(ptr, size)
- mem_Free(allocation)
- mem_Heap()
- mem_Set(ptr, char, size)
- mem_Copy(source, destination, count)
- mem_Compare(ptr1, ptr2, count)

2.20 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
- pin_HI(pin)
- pin_HI(pin)
- pin_LO(pin)
- pin_Read(pin)
- bus_In()
- bus_Out("var")
- bus_Set("var")
- bus_Write("var")
- bus_Read("var")

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	Picaso has limited but powerful I/O. There are pre-defined constants for **mode** and **pin**:	

Pin constants	Pin number on the Picaso chip	Remarks
IO1_PIN	pin 1	
IO2_PIN	pin 64	
IO3_PIN	pin 63	
IO4_PIN	pin 62	also used for BUS_RD
IO5_PIN	pin 44	also used for BUS_WR
BACKLITE	Back-light control pin.	Used internally. Permanently set as Output. **HIGH:** BACKLITE ON **LOW :** BACKLITE OFF
AUDIO_ENABLE	Amplifier Chip control pin.	Used internally. Permanently set as Output **HIGH:** Amplifier OFF **LOW :** Amplifier ON

mode constants	**mode** value	meaning	IO1	IO2	IO3	IO4	IO5
OUTPUT	0	Pin is set to an output	YES	YES	YES	YES	YES
INPUT	1	Pin is set to an input	YES	YES	YES	YES	YES

Example

```
pin_Set(OUTPUT, IO2_PIN);  // set IO2 to be used as an output
pin_Set(INPUT, IO1_PIN);   // set IO1 to be used as an 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_PIN);  // 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_PIN);  // 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 of the pin that was previously selected as an Input. Returns a "Low" (logic 0) or "High" (logic 1).	
Example	`if(pin_Read(IO1_PIN) == 1)  // read the value on IO1` `  calc_Threshold();` `else` `  ...`	

2.1.5 bus_In()

Syntax	**bus_In();**	
Arguments	**none**	
Returns	**value**	
	value	Returns the state of the bus as an 8bit value.
Description	Returns the state of the bus as an 8bit value in to the lower byte of the assigned variable. Note: The BUS_RD and BUS_WR pins are not affected.	
Example	`var1 := bus_In();` The lower byte of var1 will get loaded with the state of the bus.	

2.1.6 bus_Out(arg)

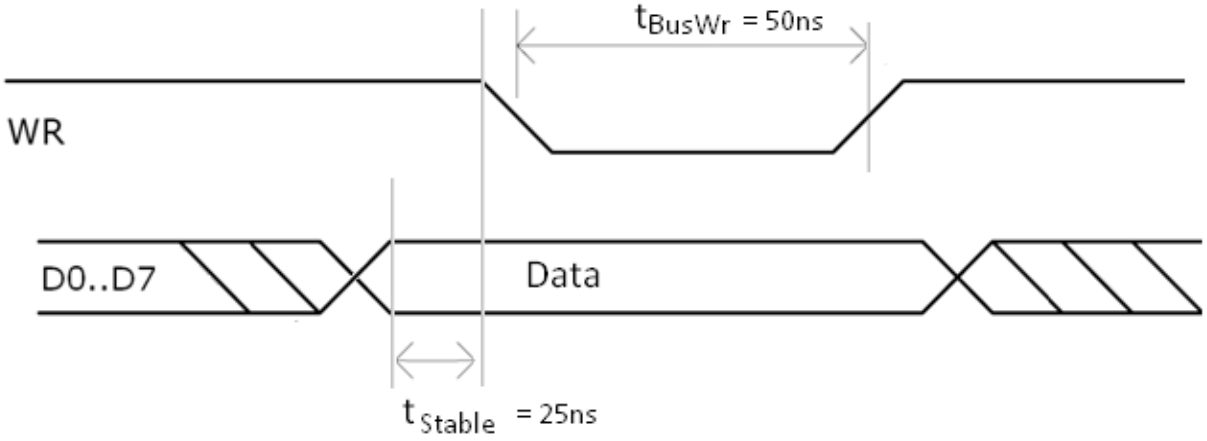
Syntax	**bus_Out(arg1);**	
Argument	**arg**	
	arg	A value (usually a constant) specifying the pin number.
	The arguments can be a variable, array element, expression or constant.	
Returns	**Nothing**	
Description	The lower byte of the argument is placed on the 8bit wide bus. The upper byte of the argument is ignored. Note: The BUS_RD and BUS_WR pins are not affected. Any BUS pins that are set to inputs are not affected.	
Example	`Var temp;` `temp := 0x0015;` `bus_Out(temp);  // Set the Bus output`	

2.1.7 bus_Set(arg)

Syntax	**bus_Set(arg1);**	
Arguments	**arg**	
	arg	A value (usually a constant) specifying the pin number. '1' sets a pin to be an input '0' sets a pin to be output.
	The arguments can be a variable, array element, expression or constant.	
Returns	**Nothing**	
Description	The lower 8 bits of arg1 are placed in the BUS direction register. a '1' sets a pin to be an input, a '0' sets a pin to be output. The upper 8 bits of arg1 are ignored. The BUS_RD and BUS_WR pins are not affected.	
Example	`var arg1;` `arg1 := 0xAA;  //` `bus_Set(arg1);  // Set the bus to value specified to arg1`	

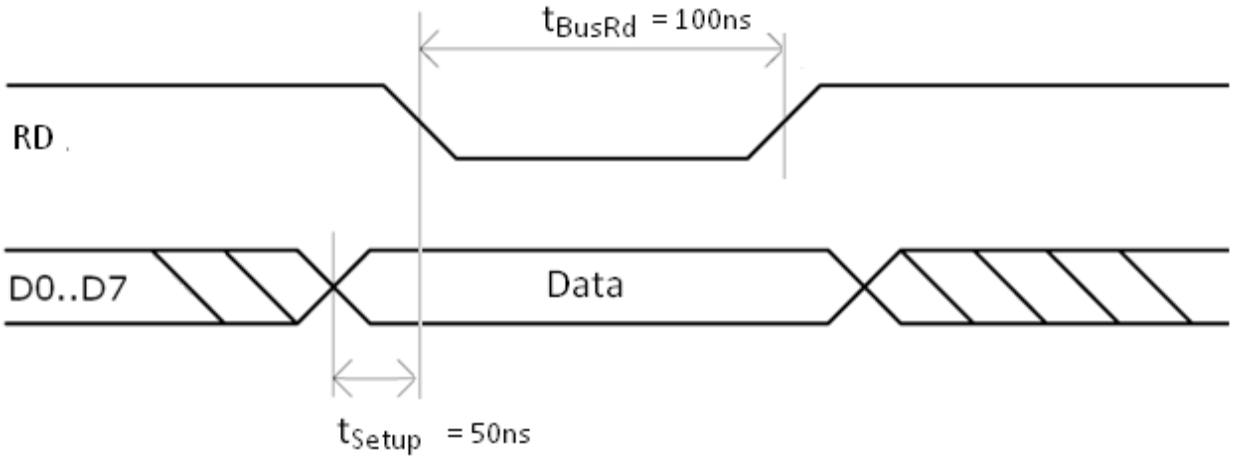
2.1.8 bus_Write(data)

Syntax	**bus_Write(data);**	
Arguments	**data**	
	data	The lower 8 bits of ***data*** are sent to the bus.
	The argument can be a variable, array element, expression or constant.	
Returns	**nothing**	
Description	The lower 8 bits of arg1 are placed on the BUS, then, after a settling delay of approx 50nsec, the BUS_WR pin is strobed LO for approx 50nsec then set back HI. The upper 8 bits of arg1 are ignored. Note: The BUS_WR pin is automatically pre-set to an output to ensure BUS write integrity.	
Example	`var data1 ;` `data1 := 0x05;` `bus_Write(data1);`	



2.1.9 bus_Read()

Syntax	**bus_Read();**	
Arguments	**none**	
Returns	**value**	
	value	Returns the state of the bus as an 8bit value.
Description	Returns the state of the bus as an 8bit value in to the lower byte of the assigned variable. Note: The BUS_RD and BUS_WR pins are not affected. The BUS_RD pin set to LO, then, after a settling delay of approx 50nsec, the BUS is read into the lower 8 bits of the assigned variable (the upper 8 bits being set to 0) the BUS_RD pin is then set back to a HI level. The BUS_RD pin is automatically pre-set to an output to ensure BUS write integrity.	
Example	`var1 := bus_Read();` The lower byte of var1 will get loaded with the state of the bus.	



2.2. System Memory Access Functions

Summary of Functions in this section:

- peekW(address)
- pokeW(address, word_value)

2.2.1 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**.	

2.2.2 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.	

2.3. 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 ()
- umul_1616(&res32, val1, val2)
- uadd_3232(&res32, &val1, &val2)
- usub_3232(&res32, &val1, &val2)
- ucmp_3232(&val1, &val2)

2.3.1 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**.	

2.3.2 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**.	

2.3.3 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**.	

2.3.4 SWAP(&var1, &var2)

Syntax	**SWAP(&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.	

2.3.5 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**.	

2.3.6 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**.	

2.3.7 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.	

2.3.8 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.	

2.3.9 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.	

2.3.10 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	

2.3.11 CY()

Syntax	**CY();**	
Arguments	**none**	
Returns	**Status**	
	Status	Returns Status of carry, 0 or 1.
Description	This function returns the carry status of an unsigned overflow from any 16 or 32bit additions or subtractions.	
Example	`var myvar;` `myvar := 0xFFF8 + 9;  // result = 1` `print("myvar ", myvar,"\nCarry ", CY(),"\n");  // carry = 1` This example will print **myvar 1** **Carry 1**	

2.3.12 umul_1616(&res32, val1, val2)

Syntax	**umul_1616(&res32, val1, val2);**	
Arguments	**&res32, val1, val2**	
	&res32	Points to 32bit result register.
	val1	16bit register or constant
	val2	16bit register or constant
Returns	**Pointer**	
	Pointer	Returns a pointer to the 32bit result. Carry and overflow are not affected.
Description	Performs an unsigned multiply of 2 x 16bit values placing the 32bit result in a 2 word array.	
Example	`var val32[2];` `var p;` `umul_1616(val32, 500, 2000);` `p := str_Ptr(val32);` `str_Printf(&p, "%ld");` This example prints 1000000	

2.3.13 uadd_3232(&res32, &val1, &val2)

Syntax	**uadd_3232(&res32, &val1, &val2);**	
Arguments	**&res32, &val1, &val2)**	
	&res32	Points to 32bit result register.
	&val1	points to 32bit augend
	&val2	points to 32bit addend
Returns	**Value**	
	Value	Returns 1 on 32bit unsigned overflow (carry). Carry flag is also set on 32bit unsigned overflow and can be read with the CY() function.
Description	Performs an unsigned addition of 2 x 32bit values placing the 32bit result in a 2 word array.	
Example	`var carry, valA[2], valB[2], Result[2];` `var p;` `valA[0] := 0;` `valA[1] := 1;` `valB[0] := 0;` `valB[1] := 1;` `carry := uadd_3232(Result, valA, valB);` `p := str_Ptr(Result);` `print("0x");` `str_Printf(&p, "%lX");  //prints the value at pointer in Hex long format.` This example will print 0x20000	

2.3.14 usub_3232(&res32, &val1, &val2)

Syntax	**usub_3232(&res32, &val1, &val2);**	
Arguments	**&res32, &val1, &val2**	
	&res32	Points to 32bit result register.
	&val1	points to 32bit augend
	&val2	points to 32bit addend
Returns	**Value**	
	Value	Returns 1 on 32bit unsigned overflow (carry). Carry flag is also set on 32bit unsigned overflow and can be read with the CY() function.
Description	Performs an unsigned subtraction of 2 x 32bit values placing the 32bit result in a 2 word array.	
Example	`var carry, valA[2], valB[2], Result[2];` `var p;` `valA[0] := 0;` `valA[1] := 0xFFFF;` `valB[0] := 0;` `valB[1] := 0xEFFF;` `carry := usub_3232(Result, valA, valB);` `p := str_Ptr(Result);` `print("0x");` `str_Printf(&p, "%lX");` `repeat forever` This example will print 0x10000000	

2.3.15 ucmp_3232(&val1, &val2)

Syntax	**ucmp_3232(&val1, &val2);**	
Arguments	**&val1, &val2**	
	&val1	points to 32bit augend
	&val2	points to 32bit addend
Returns	**Value**	
	Value	0 if equal 1 if val1 > val2 -1 if val1 < val2 This function does not affect the carry flag.
Description	Performs an unsigned comparison of 2 x 32bit values. The result of the subtraction is returned.	
Example	`var carry, valA[2], valB[2], Result;` `valA[0] := 0;` `valA[1] := 0xFFFF;` `valB[0] := 0;` `valB[1] := 0xEFFF;` `Result := cmp_3232(valA, valB); //val1 > val2` `print(Result);` `repeat forever` This example will print 1.	

2.4. 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)
  - txt_Wrap

2.4.1 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 text cursor 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.	

2.4.2 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`	

2.4.3 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 than **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
```

2.4.4 putnum(format, value)

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

Number formatting bits supplied by format

```
bit 15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0
    |  |  |  |   \___ ___/  \__ __/  \_____ _____/
    |  |  |  |       V         V           V
    |  |  |  |       |         |           |
    |  |  |  |       |         |
    |  |  |  |  (nb 0 = 16)    |           |____BASE (usually 2,10 or 16)
    |  |  |  |   digit count   |
    |  |  |  |                 |___reserved
    |  |  |  |
    |  |  |  |
    |  |  |  |
    |  |  |  |
    |  |  |  |
    |  |  |  |_____ 1 = leading zeros included
    |  |  |        0 = leading zeros suppressed
    |  |  |
    |  |  |
    |  |  |_______ 1 = leading zero blanking
    |  |
    |  |_____ sign bit (0 = signed, 1 = unsigned)
    |
     |_____ 1 = space before unsigned number
```

Pre-Defined format constants quick reference

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

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

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

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

Example

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

2.4.5 print(...)

Syntax	**print(...);**
Arguments	**See Description**
Returns	**nothing**
Description	**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 standard (word) 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).
Example	`#platform "uOLED-32028-P1_GFX2"` `//////////////////////` `// DATA STATEMENT  //` `//////////////////////` `#DATA` `  word myData` `  myString1,  Bert,  Fred,  main,  myString2,  baud,  barney, 0x1111,0x2222,0x3333,0x4444` `  byte myString1 "Data String OK\n\n",0`

```
    byte myString2 "\"(and forward referenced!)\"\n\n",0
    word baud 150,300,600,1200,2400,9600
#END

// this constant is a forward reference
#constant barney 9876

func Fred(var str)
    print("string = ", [STR] str);
endfunc

func Bert(var p1, var p2, var p3)
    print("hello from Bert\np1=",p1,"\np2=",p2, "\np3=",p3,"\n");
    return "Bert was here\n";
endfunc

func main()
    var fn;         // a variable for a handle for the function

    txt_Set(FONT_ID, FONT1);

    fn := myData[1]; //Get function pointer from data statement index
    print( [STR] fn(100,200,300) );
    // use it in a statement to prove engine ok

    fn := myData[2]; //Get function pointer from data statement index
    fn("ABC\n");     // execute the function

    // just shows where main lives
    print("\naddress of main = code[", myData[3],"]\n\n");
    // remember - a var can be a handle, variable, pointer or vector
    print( [STR] myData[0]);   // pointer table data reference
    print( [STR] myData[4]);

    repeat forever

endfunc
```

2.4.6 to(outstream)

Syntax	**to(outstream);**		
Arguments	**outstream**		
	outstream	A variable or constant specifying the destination for the **putch**, **putstr, putnum, print** and **str_Printf** functions.	
	Predefined Name	**Constant**	**putch(), putstr(), putnum(), print(), str_Printf redirection**
	APPEND	0x0000	Output is appended to user array if previous redirection was to an array.
	TEXT	0xF801	Output is directed to the **screen** (default).
	DSK	0xF802	Output is directed to the most recently open file that has been opened in write mode.
	COM0	0xFF04	Output is redirected to the **COM0** (default serial) port.
	COM1	0xFF08	Output is redirected to the **COM1** (auxilliary serial) port.
	I2C	0xF820	Output is directed to the **I2C** port.
	MDA	0xFF40	Output is directed to the **SD/SDHC** or **FLASH** media. Warning – be careful writing to a FAT16 formatted card without checking legal partitioned are else the disk formatting will be destroyed.
	(memory pointer)	Array address	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 'streams', eg – **COM0** or **COM1**, an open FAT16 file with **DSK**, to raw media with **MDA** (media), or to the I2C port with **I2C**. 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, print** or **str_Printf** has completed its action. The **APPEND** argument is used to append 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
```

2.4.7 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.
```

2.4.8 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()`	

2.4.9 strwidth(pointer)

Syntax	**strwidth(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()`	

2.4.10 strheight()

Syntax	**strheight();**	
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()`	

2.4.11 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()`	

2.4.12 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 or FONT1 = system font 2 0r FONT3 = Default fonts Note: The value could be the name of a custom font included in a users program in a data statement. See examples in the 4DGL Workshop3 IDE.	0 or FONT1 1 or FONT2 2 or FONT3
3	**TEXT_WIDTH**	Set the text width multiplier.	1 to 16 (Default =1)
4	**TEXT_HEIGHT**	Set the text height multiplier.	1 to 16 (Default =1)
5	**TEXT_XGAP**	Set the pixel gap between characters. The gap is in pixel units	0 to 32(Default =0)
6	**TEXT_YGAP**	Set the pixel gap between lines. The gap is in pixel units.	0 to 32(Default =0)
7	**TEXT_PRINTDELAY**	Set the delay between character printing	(Default 0msec)
8	**TEXT_OPACITY**	Selects whether or not the 'background' pixels are drawn (default mode is OPAQUE)	0 or **TRANSPARENT** 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**
14	**TEXT_WRAP**	Sets the pixel position where text wrap will occur at RHS The feature automatically resets when screen mode is changed. The value is in pixel units. Default value is 0.	0 to n(**OFF** or Value)

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 or FONT1 = system font 2 0r FONT3 = Default fonts Note: The value could also be the name of a custom font included in a users program in a data statement, or the handle returned from file_LoadImageControl() for a uSD based font.	0 to 2 or FONT1 FONT2 FONT3
txt_Width(multiplier)	Set the text width multiplier (note #6)	1 to 16 (Default =1)
txt_Height(multiplier)	Set the text height multiplier (note #6)	1 to 16 (Default =1)
txt_Xgap(pixelcount)	Set the pixel gap between characters. The gap is in pixel units	0 to 32(Default =0)
txt_Ygap(pixelcount)	Set the pixel gap between lines. The gap is in pixel units.	0 to 32(Default =0)
txt_Delay(millisecs)	Set the delay between character printing .	(not used)
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
txt_Wrap	Sets the pixel position where text wrap will occur at RHS The feature automatically resets when screen mode is changed. The value is in pixel units. Default value is 0.	0 to n(OFF or Value)

2.5. Ctype Functions

Summary of Functions in this section:

- isdigit(char)
- isxdigit(char)
- isupper(char)
- islower(char)
- isalpha(char)
- isalnum(char)
- isprint(char)
- isspace(char)
- iswhite(char)
- toupper(char)
- tolower(char)
- LObyte(var)
- HIbyte(var)
- ByteSwap(var)

2.5.1 isdigit(char)

Syntax	**isdigit(char);**	
Arguments	**char**	
	char	Specifies the ASCII character for the test.
Returns	**Status**	
	Status	**0**: Character is not as ASCII digit **1**: Character is an ASCII digit.
Description	Tests the character parameter and returns a 1 if the character is an ascii digit else returns a 0. Valid range : "0123456789".	
Example		

```
func main()
    var ch;
    var stat;
    gfx_Cls();
    txt_Set(FONT_ID, FONT2);
    print ("Serial Input Test\n");
    print ("Download prog to flash\n");
    print ("Then use debug terminal\n");

    to(COM0); print("serial input test:\n");

    // now just stay in a loop
    repeat

    ch := serin();
    if (ch != -1)
      print( [CHR] ch );   // if a key was received from PC,
                           // print its ascii value
      if (isdigit(ch)) print("Character is an ASCII digit");
      if (isxdigit(ch)) print("Character is ASCII Hexadecimal");
      if (isupper(ch)) print("Character is ASCII uppercase letter");
      if (islower(ch)) print("Character is ASCII uppercase letter");
      if (isalpha(ch)) print("Character is an ASCII uppercase or
                                                        lowercase");
      if (isalnum(ch)) print("Character is an ASCII Alphanumeric");
      if (isprint(ch)) print("Character is a printable ASCII");
      if (isspace(ch)) print("Character is a space type character");

    endif

    forever

endfunc;
```

2.5.2 isxdigit(char)

Syntax	**isxdigit(char);**	
Arguments	**char**	
	char	Specifies the ASCII character for the test.
Returns	**Status**	
	Status	**0**: Character is not as ASCII hexadecimal digit **1**: Character is an ASCII hexadecimal digit.
Description	Tests the character parameter and returns a 1 if the character is an ascii hexadecimal digit else returns a 0. Valid range : "0123456789ABCDEF".	
Example	`Refer to Sec 2.5.1`	

2.5.3 isupper(char)

Syntax	**isupper(char);**	
Arguments	**char**	
	char	Specifies the ASCII character for the test.
Returns	**Status**	
	Status	**0**: Character is not an ASCII upper case letter. **1**: Character is an ASCII upper case letter.
Description	Tests the character parameter and returns a 1 if the character is an ASCII upper case letter else returns a 0. Valid range : "ABCDEF....WXYZ".	
Example	`Refer to Sec 2.5.1`	

2.5.4 islower(char)

Syntax	**islower(char);**	
Arguments	**char**	
	char	Specifies the ASCII character for the test.
Returns	**Status**	
	Status	**0**: Character is not an ASCII lower case letter **1**: Character is an ASCII lower case letter.
Description	Tests the character parameter and returns a 1 if the character is an ASCII lower case letter else returns a 0. Valid range : "abcd....wxyz".	
Example	`Refer to Sec 2.5.1`	

2.5.5 isalpha(char)

Syntax	**isalpha(char);**	
Arguments	**char**	
	char	Specifies the ASCII character for the test.
Returns	**Status**	
	Status	**0**: Character is not as ASCII lower or upper case letter. **1**: Character is an ASCII lower or upper case letter..
Description	Tests the character parameter and returns a 1 if the character is an ASCII lower or upper case letter else returns a 0. Valid range : "abcd....wxyz", “ABCD....WXYZ”	
Example	`Refer to Sec 2.5.1`	

2.5.6 isalnum(char)

Syntax	**isalnum(char);**	
Arguments	**char**	
	char	Specifies the ASCII character for the test.
Returns	**Status**	
	Status	**0**: Character is not as ASCII Alphanumeric character. **1**: Character is an ASCII Alphanumeric character.
Description	Tests the character parameter and returns a 1 if the character is an ASCII Alphanumeric else returns a 0. Valid range : "abcd....wxyz", “ABCD....WXYZ”, “0123456789”	
Example	`Refer to Sec 2.5.1`	

2.5.7 isprint(char)

Syntax	**isprint(char);**	
Arguments	**char**	
	char	Specifies the ASCII character for the test.
Returns	**Status**	
	Status	**0**: Character is not a printable ASCII character. **1**: Character is a printable ASCII character.
Description	Tests the character parameter and returns a 1 if the character is a printable ASCII character else returns a 0. Valid range : 0x20... 0x7F	
Example	`Refer to Sec 2.5.1`	

2.5.8 isspace(char)

Syntax	**isspace(char);**	
Arguments	**char**	
	char	Specifies the ASCII character for the test.
Returns	**Status**	
	Status	**0**: Character is not a space type character. **1**: Character is a space type character.
Description	Tests the character parameter and returns a 1 if the character is any one of the space type character else returns a 0. Valid range : space, formfeed, newline, carriage return, tab, vertical tab.	
Example	`Refer to Sec 2.5.1`	

2.5.9 toupper(char)

Syntax	**toupper(char);**	
Arguments	**char**	
	char	Specifies the ASCII character for the test.
Returns	**Char**	
	Char	**"ABCD...WXYZ"**: If character is lower case letter. **char**: If character is not a lower case letter..
Description	Tests the character parameter and if the character is a lower case letter it returns the upper case equivalent else returns the passed char. Valid range : "abcd ... wxyz".	
Example		

```
func main()
    var ch, Upconvch, Loconvch;
    var stat;
    gfx_Cls();
    txt_Set(FONT_ID, FONT2);
    print ("Serial Input Test\n");
    print ("Download prog to flash\n");
    print ("Then use debug terminal\n");

    to(COM0); print("serial input test:\n");

    // now just stay in a loop
    repeat

    ch := serin();
    if (ch != -1)
      print( [CHR] ch );   // if a key was received from PC,
                           // print its ascii value

    if (isupper(ch))
          print("Uppercase ASCII found. Converting to lowercase");
        Loconvch := tolower(ch);
      endif
      if (islower(ch))
         print("Lowercase ASCII found. Converting to Uppercase");
         Upconvch := toupper(ch);
      endif

    endif

    forever

endfunc;
```

2.5.10 tolower(char)

Syntax	**tolower(char);**	
Arguments	**char**	
	char	Specifies the ASCII character for the test.
Returns	**Status**	
	Status	**"abcd...wxyz"**: If character is upper case letter. **char**: If character is not a upper case letter...
Description	Tests the character parameter and if the character is a lower case letter it returns the upper case equivalent else returns the passed char. Valid range : "ABCD ... WXYZ".	
Example	`Refer to Sec 2.5.9`	

2.5.11 LObyte(var)

Syntax	**LObyte(var);**	
Arguments	**var**	
	var	User variable.
Returns	**byte**	
	byte	Returns the lower byte (lower 8 bit) of a 16 bit variable.
Description	Returns the lower byte (lower 8 bit) of a 16 bit variable.	
Example	`myvar := LObyte(myvar2);`	

2.5.12 HIbyte(var)

Syntax	**Hibyte(var);**	
Arguments	**var**	
	var	User variable.
Returns	**byte**	
	byte	Returns the upper byte (upper 8 bits) of a 16 bit variable.
Description	Returns the upper byte (upper 8 bits) of a 16 bit variable.	
Example	`myvar := HIbyte(myvar2);`	

2.5.13 ByteSwap(var)

Syntax	**ByteSwap(var);**	
Arguments	**var**	
	var	User variable.
Returns	**Status**	
	Status	Returns the endian swapped value of a 16 bit variable.
Description	Returns the swapped upper and lower bytes of a 16 bit variable.	
Example	`myvar := ByteSwap(myvar2);`	

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_Ellipse(x, y, xrad, yrad, colour)
- gfx_EllipseFilled(x, y, xrad, yrad, colour)
- gfx_Button(state, x, y, buttonColour, textColour, font, textWidth, textHeight, text)
- gfx_Panel(state, x, y, width, height, colour)
- gfx_Slider(mode, x1, y1, x2, y2, colour, scale, value)
- gfx_ScreenCopyPaste(xs, ys, xd, yd, width, height)
- gfx_RGBto565(RED, GREEN, BLUE)
- gfx_332to565(COLOUR8BIT)
- gfx_TriangleFilled(x1, y1, x2, y2, x3, y3, colr)
- gfx_PolygonFilled(n, &vx, &vy, colr)
- gfx_Origin(x, y)
- gfx_Get(mode)
- gfx_ClipWindow(x1, y1, x2, y2)
- gfx_Set(function, value)

  gfx_Set shortcuts:

  - gfx_PenSize(mode)
  - gfx_BGcolour(colour)
  - gfx_ObjectColour(colour)
  - gfx_Clipping(mode)
  - gfx_TransparentColour(colour)
  - gfx_Transparency(mode)
  - gfx_FrameDelay(delay)
  - gfx_ScreenMode(delay)
  - gfx_OutlineColour(colour)
  - gfx_Contrast(value)

- gfx_LinePattern(pattern)
- gfx_ColourMode(mode)
- gfx_BevelWidth(mode)
- gfx_BevelShadow(value)
- gfx_Xorigin(offset)
- gfx_Yorigin(offset)

2.6.1 gfx_Cls()

Syntax	**gfx_Cls();**
Arguments	**none**
Returns	**nothing**
Description	Clear the screen using the current background colour. gfx_Cls() command brings some of the settings back to default; such as, • Transparency turned OFF • Outline colour set to BLACK • Opacity set to OPAQUE • Pen set to OUTLINE • Line patterns set to OFF • Right text margin set to full width • Text magnifications set to 1 • All origins set to 0:0 The alternative to maintain settings and clear screen is to draw a filled rectangle with the required 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, RED);` 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	`if(state ==  TOUCH_RELEASED)  // if there's a release;` `  gfx_CircleFilled(x, y, 10, RED); // we'll draw a solid red circle` `                                 // of radius=10 on touch release` `endif`	

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, RED);` 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, RED);` 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, RED);` 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	`gfx_Rectangle(10, 10, 30, 30, GREEN);` 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_RectangleFilled(30,30,80,80, RED);` This example draws a filled RED rectangle 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,CYAN);` 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(0xRED);` `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	`gfx_MoveTo(30, 30);` `gfx_Bullet(10); // Draw a 10pixel radius Bullet at x=30, y=30.`	

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		

```
#inherit "4DGL_16bitColours.fnc"

func help()
    var x, y, state;

    print("TOUCHE ME");

    touch_Set(TOUCH_ENABLE);      // lets enable the touch screen
    while(touch_Get(TOUCH_STATUS) != TOUCH_PRESSED); //Wait for touch

    // we'll need a place on the screen to start with
    gfx_MoveTo(touch_Get( TOUCH_GETX), touch_Get( TOUCH_GETY));

    gfx_Set(OBJECT_COLOUR, WHITE); // this will be our line colour

    while(1)
       state := touch_Get(TOUCH_STATUS); // Look for touch activity
       x := touch_Get(TOUCH_GETX); // Grab x and the
       y := touch_Get(TOUCH_GETY); // y coordinates of the touch

       if(state == TOUCH_PRESSED)    // if there's a press
          gfx_LineTo(x, y); // Draw a line from previous spot
       endif

       if(state ==  TOUCH_RELEASED)        // if there's a release;
          gfx_CircleFilled(x, y, 10, RED);// Draw a solid red circle
       endif

       if(state == TOUCH_MOVING)           // if there's movement
         gfx_PutPixel(x, y, LIGHTGREEN);  // we'll draw a green pixel
       endif
    wend                                   // Repeat forever
endfunc
```

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.

2.6.27 gfx_Ellipse(x, y, xrad, yrad, colour)

Syntax	**gfx_Ellipse(x, y, xrad, yrad, colour);**	
Arguments	**x, y, xrad, yrad, colour**	
	x, y	specifies the horizontal and vertical position of the centre of ellipse
	xrad, yrad	Specifies x-radius and y-radius of the ellipse.
	colour	Specifies the colour for the lines
	The arguments can be a variable, array element, expression or constant	
Returns	**nothing**	
Description	Plots a coloured Ellipse on the screen at centre x,y with xradius = xrad and yradius = yrad. if PenSize = 0 Ellipse is Solid if PenSize = 1 Ellipse is Outline	
Example	`gfx_Ellipse(200,80,5,10,YELLOW);`	

2.6.28 gfx_EllipseFilled(x, y, xrad, yrad, colour)

Syntax	**gfx_EllipseFilled(x, y, xrad, yrad, colour);**	
Arguments	**x, y, xrad, yrad, colour**	
	x, y	specifies the horizontal and vertical position of the centre of ellipse
	xrad, yrad	Specifies x-radius and y-radius of the ellipse.
	colour	Specifies the colour for the lines
	The arguments can be a variable, array element, expression or constant	
Returns	**nothing**	
Description	Plots a solid coloured Ellipse on the screen at centre x,y with xradius = xrad and yradius = yrad.	
Example	`gfx_EllipseFilled(200,110,10,5,GREEN);`	

2.6.29 gfx_Button(state, x, y, buttonColour, txtColour, font, txtWidth txtHeight, text)

Syntax	**gfx_Button(state, x, y, buttonColour, txtColour, font, txtWidth, txtHeight, text);**	
Arguments	**state, x, y, buttonColour, txtColour, font, txtWidth, txtHeight, text**	
	state	0 = Button depressed; 1 = Button raised.
	x, y	Specifies the top left corner position of the button on the screen.
	buttonColour	Button colour
	txtColour	Text Colour
	font	Specifies the Font ID.
	txtWidth	specifies the width of the text. This value is the font width multiplier and minimum value must be 1.
	txtHeight	specifies the height of the text. This value is the font height multiplier and minimum value must be 1.
	text	Specifies the text string. The text string must be within the range of printable ascii character set. The string may have \n characters embedded to create a multiline button.
Returns	**nothing**	
Description	Draws a 3 dimensional Text Button at screen location defined by x, y parameters (top left corner). The size of the button depends on the font, width, height and length of the text. The button can contain multiple lines of text by having the \n character embedded in the string for the end of line marker. In this case, the widest text in the string sets the overall width, and the height of the button is set by the number of text lines. In the case of multiple lines, each line is left justified. If you wish to centre or right justify the text, you will need to prepare the text string according to your requirements.	
Example		

```
#constant LEFT 30
#constant TOP  150
#constant TEXTWIDTH  2
#constant TEXTHEIGHT 2

//---------------------------------------------------------------------
func main()

// Draw a button as a Text Box (indented)
gfx_Button(DOWN, 0, 30, GREEN, WHITE, FONT4, TEXTWIDTH, TEXTHEIGHT,
"4DGL-Demo");

touch_Set(TOUCH_ENABLE);

repeat
        // Draw the Push Button (raised)
        gfx_Button(UP, LEFT, TOP, BLUE, RED, FONT4, TEXTWIDTH,
      TEXTHEIGHT, " PRESS ");
      // set touch detect region to that of the push button
      touch_DetectRegion(LEFT, TOP, gfx_Get(RIGHT_POS),
      gfx_Get(BOTTOM_POS));

      // Wait until the button is pressed
      while(touch_Get(TOUCH_STATUS) != TOUCH_PRESS);
```

```
        // now redraw the Push Button (depressed)
        gfx_Button(DOWN, LEFT, TOP, BLUE, WHITE, FONT4, TEXTWIDTH,
        TEXTHEIGHT, " PRESS ");

        // Wait until the button is pressed
        while(touch_Get(TOUCH_STATUS) != TOUCH_RELEASE);
forever

endfunc
```

2.6.30 gfx_Panel(state, x, y, Width, Height, Colour)

Syntax	**gfx_Panel(state, x, y, Width, Height, Colour);**	
Arguments	**state, x, y, buttonColour, txtColour, font, txtWidth, txtHeight, text**	
	state	0 = recessed; 1 = raised.
	x, y	Specifies the top left corner position of the panel on the screen.
	Width	specifies the width of the panel.
	Height	Specifies the Height of the panel.
	Colour	Specifies the colour of the panel.
Returns	**nothing**	
Description	Draws a 3 dimensional rectangular panel at a screen location defined by x, y parameters (top left corner). The size of the panel is set with the width and height parameters. The colour is defined by colour The state parameter determines the appearance of the panel, 0 = recessed, 1 = raised.	
Example		

```
#constant LEFT 15
#constant TOP  15
#constant WIDTH  100
#constant HEIGHT 100

func main()

    // Draw a panel
    gfx_Panel(RAISED,  LEFT, TOP, WIDTH, HEIGHT, GRAY);

    repeat forever

endfunc
```

2.6.31 gfx_Slider(mode, x1, y1, x2, y2, colour, scale, value)

Syntax	**gfx_Slider(mode, x1, y1, x2, y2, colour, scale, value);**	
Arguments	**mode, x1, y1, x2, y2, colour, scale, value**	
	mode	mode = 0 : Slider Indented, mode = 1 : Slider Raised, mode 2, Slider Hidden (background colour).
	x1, y1	specifies the top left corner position of the slider on the screen.
	x2, y2	specifies the bottom right corner position of the slider on the screen.
	colour	specifies the colour of the Slider bar.
	Scale	scale = n : sets the full scale range of the slider for the thumb from 0 to n.
	Value	if value positive, sets the relative position of the thumb on the slider bar, else set thumb to ABS position of the negative number.
Returns	**If the value parameter was a positive number (i.e:- value is a proportion of the scale parameter) , the true (implied x or y axis) position of the thumb is returned.** **If the value parameter was a negative number (i.e:- thumb is being set to an ABSolute graphics position) , the actual slider value (which is a proportion of the scale parameter) is returned.**	
Description	Draws a vertical or horizontal slider bar on the screen. The gfx_Slider function has several different modes of operation. In order to minimise the amount of graphics functions we need, all modes of operation are selected naturally depending on the parameter values. Selection rules: 1a] if x2-x1 > y2-y1 slider is assumed to be horizontal (ie: if width > height, slider is horizontal) 1b] if x2-x1 <= y2-y1 slider is assumed to be vertical (ie: if height <= width, slider is horizontal) 2a] If value is positive, thumb is set to the position that is the proportion of value to the scale parameter.(used to set the control to the actual value of a variable) 2b] If value is negative, thumb is driven to the graphics position set by the ABSolute of value value. (used to set thumb to its actual graphical position (usually by touch screen) 3] The thumb colour is determine by gfx_Set(OBJECT_COLOUR, value); , however, if the current object colour is BLACK, a darkened shade of the colour parameter is used for the thumb .	
Example	`func drawRedSlider()` `  gfx_Slider(0,rSlider[0],rSlider[1],rSlider[2],rSlider[3],RED,255, valR);` `  txt_MoveCursor(1,12);` `  txt_Set(TEXT_OPACITY, OPAQUE);` `  txt_Set(TEXT_COLOUR, RED);` `  print ("  ");` `  txt_MoveCursor(1,12);` `  print ([DEC] valR);` `endfunc`	

2.6.32 gfx_ScreenCopyPaste(xs, ys, xd, yd, width, height)

Syntax	**gfx_ScreenCopyPaste(xs, ys, xd, yd, width, height);**	
Arguments	**xs, ys, xd, yd, width, height**	
	xs, ys	Specifies the horizonal and vertical position of the top left corner of the area to be copied (source).
	xd, yd	Specifies the horizontal and vertical position of the top left corner of where the paste is to be made (destination).
	width	Specifies the width of the copied area.
	height	Specifies the height of the copied area.
	The arguments can be a variable, array element, expression or constant	
Returns	**nothing**	
Description	Copies an area of a screen from xs, ys of size given by width and height parameters and pastes it to another location determined by xd, yd.	
Example	`gfx_ScreenCopyPaste(10,10, 100, 100, 40, 40);` `// Copies 40x40 pixels originating from point (10,10) to (100,100);`	

2.6.33 gfx_RGBto565(RED, GREEN, BLUE)

Syntax	**gfx_RGBto565(RED, GREEN, BLUE);**	
Arguments	**RED, GREEN, BLUE**	
	RED	8bit colour value for RED.
	GREEN	8bit colour value for GREEN. .
	BLUE	8bit colour value for BLUE.
	The arguments can be a variable, array element, expression or constant	
Returns	Returns the 16bit (RED:5, GREEN:6, BLUE:5 format) colour value.	
Description	Returns the 16bit (RED:5, GREEN:6, BLUE:5 format) colour value of a 24bit (RED:8, GREEN:8, BLUE:8 format) colour.	
Example	`var colorRGB;` `colorRGB := gfx_RGBto565(170, 126, 0);` `// convert 8bit Red, Green and Blue color values to 16bit 565 color value`	

2.6.34 gfx_332to565(COLOUR)

Syntax	**gfx_332to565(COLOUR);**	
Arguments	**Colour**	
	Colour	8bit colour value. 3bits for RED, 3bits for GREEN, 2bits for BLUE.
Returns	**Returns the 16bit (RED:5, GREEN:6, BLUE:5 format) value**	
Description	Returns the 16bit (RED:5, GREEN:6, BLUE:5 format) value of an 8bit (RED:3, GREEN:3, BLUE:2 format) colour	
Example	`var color565;` `color565 := gfx_332to565(0b11010100);` `// Convert 8bit 332 color value to 16bit 565 color value`	

2.6.35 gfx_TriangleFilled(x1, y1, x2, y2, x3, y3, colour)

Syntax	**gfx_TriangleFilled(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 Solid triangle between vertices x1,y1 , x2,y2 and x3,y3 using the specified colour.	
Example	`gfx_TriangleFilled(10,10,30,10,20,30,CYAN);` This example draws a CYAN Solid triangle with vertices at 10,10 30,10 20,30	

2.6.36 gfx_PolygonFilled(n, vx, vy, colour)

Syntax	**gfx_PolygonFilled(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	Draws a solid Polygon between specified vertices: x1,y1 x2,y2 .. xn,yn using the specified colour. The last point is drawn back to the first point, completing the polygon. Vertices must be minimum of 3 and can be specified in any fashion	
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_PolygonFilled(7, vx, vy, RED);` `  repeat forever` `endfunc` This example draws a simple filled polygon	

2.6.37 gfx_Origin(x, y)

Syntax	**gfx_Origin(x, y);**	
Arguments	**x, y**	
	x, y	specifies the horizontal and vertical position of the top left corner of the clipping window.
Returns	**nothing**	
Description	Sets relative screen offset for horizontal and vertical for the top left corner for graphics objects.	
Example	`gfx_Origin(10, 20); // Sets origin position at (10,20)`	

2.6.38 gfx_Get(mode)

Syntax	**gfx_Get(mode);**	
Arguments	**mode**	
	mode	mode = 0 : Current orientations Max X Value (X_MAX) mode = 1 : Current orientations Max Y Value (Y_MAX) mode = 2 : Left location of Object mode = 3 : Top location of Object mode = 4 : Right location of Object mode = 5 : Bottom location of Object mode = 6 : Get current internal X position mode = 7 : Get current internal Y position
Returns	**Mode0** Returns the maximum horizontal value of the display. **Mode1** Returns the maximum vertical value of the display. **Mode2** Returns the left location of the last drawn object such as a slider or button or an image/video. **Mode3** Returns the top location of the last drawn object such as a slider or button or an image/video. **Mode4** Returns the right location of the last drawn object such as a slider or button or an image/video. **Mode5** Returns the bottom location of the last drawn object such as a slider or button or an image/video. **Mode6** Returns the internal X position that was set with MoveTo(x, y); or gfx_Set(X_ORG, pos); **Mode7** Returns the internal Y position that was set with MoveTo(x, y); or gfx_Set(X_ORG, pos);	
Description	Returns various graphics parameters to caller.	
Example	var := gfx_Get(X_MAX); //Returns the maximum horizontal resolution of the display var := gfx_Get(0); var := gfx_Get(Y_MAX); //Returns the maximum vertical resolution of the display var := gfx_Get(1); var := gfx_Get(RIGHT_POS); //Returns the right location of the last drawn object //that only has top, left parameters such as a button // or an image/video. var := gfx_Get(2); var := gfx_Get(BOTTOM_POS); //Returns the bottom location of the last drawn object //that only has top, left parameters such as a button //or an image/video. var := gfx_Get(3);	

2.6.39 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	

2.6.40 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**	
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**
BACKGROUND_COLOUR	Set the screen background colour	Colour, 0-65535
OBJECT_COLOUR	Generic colour for gfx_LineTo(...), gfx_LineRel(...), gfx_Dot(), gfx_Bullet(...) and gfx_BoxTo(...)	Colour, 0-65535
CLIPPING	Turns clipping on/off. The clipping points are set with **gfx_ClipWindow(...)** and must be set first.	1 or 0 (**ON** or **OFF**)
TRANSPARENT_COLOUR	Colour that needs to be made transparent.	Colour, 0-65535
TRANSPARENCY	Turn the transparency ON or OFF. Transparency is automatically turned OFF after the next image or video command.	1 or 0 (**ON** or **OFF**)
FRAME_DELAY	Set the inter frame delay for **media_Video(...)**	0 to 255msec
SCREEN_MODE	Set required screen behaviour/orientation.	0 or **LANDSCAPE** 1 or **LANDSCAPE _R** 2 or **PORTRAIT** 3 or **PORTRAIT_R**
OUTLINE_COLOUR	Outline colour for rectangles and circles (set to 0 for no effect)	Colour, 0-65535
CONTRAST	**OLED MODULES:** Set contrast value, 0 = display off, 1-9 = contrast level **LCD MODULES:** contrast 0 = display OFF, non-zero = display ON **EXCEPTION:** uLCD-43 supports Contrast values from 1-15 and 0 to turn the Display off. 3202X-P1 supports Contrast values from 1 to 9 and 0 to turn the Display off.	0 or **OFF** 1 to 9 for levels 1 or 0 (**ON** or **OFF**)

	Note: Does not apply to uVGA-II/III modules.	
BEVEL_WIDTH	Set Button Bevel Width, 0 pixel to 15pixels.	0 None 1 to 15 pixels
SCREEN_RES	Set VGA Screen resolution. Applies to uVGA-II/III only.	0 for 320x240 1 for 640 x 480 2 for800 x 480
DISPLAY_PAGE	Choose Page to be displayed. Value depends on the resolution set. Applies to uVGA-II/III and uLCD-43 only.	e.g. 00hex-04hex for 320x240 resolution on a uVGA-II/III.
READ_PAGE	Choose the Page to be read. Value depends on the resolution set. Applies to uVGA-II/III and uLCD-43 only.	e.g. 00hex-04hex for 320x240 resolution on a uVGA-II/III.
WRITE_PAGE	Choose the Page to be written. Value depends on the resolution set. Applies to uVGA-II/III and uLCD-43 only.	e.g. 00hex-04hex for 320x240 resolution on a uVGA-II/III.

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_TransparentColour(colour)	Colour that needs to be made transparent.	Colour, 0-65535
gfx_Transparency(mode)	Turn the transparency ON or OFF.	1 or 0 (**ON** or **OFF**)
gfx_FrameDelay(delay)	Set the inter frame delay for **media_Video(...)**	0 to 255msec
gfx_ScreenMode(mode)	Graphics orientation LANDSCAPE, LANDSCAPE_R, PORTRAIT, PORTRAIT_R	1 or **LANDSCAPE** 2 or **LANDSCAPE _R** 3 or **PORTRAIT** 4 or **PORTRAIT_R**
gfx_OutlineColour(colour)	Outline colour for rectangles and circles. (set to 0 for no effect)	Colour 0-65535
gfx_Contrast(value)	**OLED MODULES:** Set contrast value, 0 = display off, 1-9 = contrast level **LCD MODULES:** contrast 0 = display OFF, non-zero = display ON **EXCEPTION:** uLCD-43 supports Contrast values from 1-15 and 0 to turn the Display off. 3202X-P1 supports Contrast values from 1 to 9 and 0 to turn the Display off. **Note**: Does not apply to uVGA-II/III modules.	0 or **OFF** 1 to 9 for levels 1 or 0 (**ON** or **OFF**)
gfx_LinePattern(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	0 bits for pixels on 1 bits for pixels off

	turned off. See code examples for further reference.	
gfx_ColourMode(mode)	Sets 8 or 16bit colour mode Function not available, fixed as 16bit mode.	0 or **COLOUR16** 1 or **COLOUR8**
gfx_BevelWidth(mode)	graphics button bevel width	0 None 1 to 15 pixels
gfx_BevelShadow(value)	graphics button bevel shadow depth	
gfx_Xorigin(offset)	graphics X origin	
gfx_Yorigin(offset)	graphics Y origin	

2.7. Display I/O Functions

These functions allow direct display access for fast blitting operations.

Summary of Functions in this section:

- disp_SetReg(register, data)
- disp_setGRAM(x1, y1, x2, y2)
- disp_WrGRAM(colour)
- disp_WriteControl(value)
- disp_WriteWord(value)
- disp_ReadWord()
- disp_Sync(line)
- disp_Disconnect()
- disp_Init()

2.7.1 disp_SetReg(register, data)

Syntax	**disp_SetReg(register, data);**	
Arguments	**register, data**	
	register	Refer to the display driver data sheet
	data	Refer to the display driver data sheet
Returns	**nothing**	
Description	Sets the Display driver IC register.	

2.7.2 disp_setGRAM(x1, y1, x2, y2)

Syntax	**disp_setGRAM(x1, y1, x2, y2);**	
Arguments	**x1, y1, x2, y2**	
	x1, y1	Top left of the GRAM window.
	x2, y2	Bottom right of the GRAM window.
Returns	**the LO word of the 32 bit pixel count is returned**	
Description	Prepares the GRAM area for user access. The lower 16bits of the pixel count in the selected area is returned This is usually all that is needed unlse GRAM area exceeds 256^2. A copy of the 32bit value can be found in GRAM_PIXEL_COUNT_LO and GRAM_PIXEL_COUNT_HI.	
Example	`disp_setGRAM(40, 60, 100, 150);`	

2.7.3 disp_WrGRAM(colour)

Syntax	**disp_WrGRAM(colour);**	
Arguments	**colour**	
	colour	Pixel color to be populated.
Returns	**nothing**	
Description	Data can be written to the GRAM consecutively using this function once the GRAM access window has been setup.	
Example	`disp_WrGRAM(0xFFF0);`	

2.7.4 disp_WriteControl(value)

Syntax	**disp_WriteControl(value);**	
Arguments	**value**	
	value	Specifies the 16 bit value to be written to the display control register.
	The arguments can be a variable, array element, expression or constant	
Returns	**nothing**	
Description	Sends a 16 bit 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	`disp_WriteControl(0x0FFA);`	

2.7.5 disp_WriteWord(value)

Syntax	**disp_WriteWord(value);**	
Arguments	**value**	
	value	Specifies the value to be written to the display data register.
	The arguments can be a variable, array element, expression or constant	
Returns	**nothing**	
Description	Sends a 16 bit 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	`disp_WriteWord(0x7FF0);`	

2.7.6 disp_ReadWord(value)

Syntax	**disp_ReadWord(value);**	
Arguments	**value**	
	value	Specifies the value to be read from the display data register.
Returns	**Returns 16 bit value in the register.**	
Description	Read a word from the display.	
Example	`var val;` `val := disp_ReadWord();`	

2.7.7 disp_Sync(line)

Syntax	**disp_Sync(line);**	
Arguments	**line**	
	line	Scan line.
Returns	**Returns 16 bit value in the register.**	
Description	Allows the program to synchronise writing to the hardware for flicker free operation. Some experimentation may be needed to find an optimum line for disp_Sync depending on the graphics operation. The higher the value, the slower the throughput. A certain point will be reached (number of scanlines + blanking lines within the vertical retrace period) where it will just 'hang up' stopping the entire process. Eg, in 640x480 mode, if the 'lines' value is 507, operation will be slowest (as its actually right at the end of the blanking period) and 508 will cause a hangup situation as it is above the highest scanline value. **Note:** Applies to uVGA-II/III modules only.	

2.7.8 disp_Disconnect()

Syntax	**disp_Disconnect();**
Arguments	**none**
Returns	**nothing**
Description	This function disconnects the display driver pins and/or reconfigures it to achieve its lowest possible power consumption. Use after disabling peripheral power to ensure the minimal power usage by the display. Disp_Init() should be used to reinitialise the display. New in v3.8 PmmC

2.7.9 disp_Init()

Syntax	**disp_Init();**
Arguments	**none**
Returns	**nothing**
Description	This function is used to initialise the display. This is useful in a number of situations, however mainly for the uLCD-xx-PTU modules which have the ability to disable the power supply to the display for low power sleep modes. This function is required to re-initialise the display once power to the display has been restored, so the display is usable once again.

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 Picaso SPI port.

Summary of Functions in this section:

- media_Init()
- media_SetAdd(HIword, LOword)
- media_SetSector(HIword, LOword)
- media_RdSector(Destination_Address)
- media_WrSector(Source_Address)
- 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 Picaso 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	**result**	
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_RdSector(Destination_Address)

Syntax	**media_RdSector(Destination_Address);**	
Arguments	**Destination_Address**	
	Destination_Address	Destination block pointed to by the internal Sector pointer.
	The argument must be a pointer to an array of size 256 words for the sector data which will be 512 bytes	
Returns	Returns TRUE if media response was TRUE. Returns 512 bytes (256 words) in to a destination block.	
Description	Reads and Returns 512 bytes (256 words) into a destination block (eg rdblock[256]) pointed to by the internal Sector pointer. After the read the Sector pointer is automatically incremented by 1.	
Example	`var rdblock[256];` `media_SetSector(0,10)` `if (media_RdSector(rdblock));` `Print("Data collected");` `endif` This example sets a 512 bytes block and collects data from the address pointed to by media_SetSector command.	

2.8.5 media_WrSector(Source_Address)

Syntax	**media_WrSector(Source_Address);**	
Arguments	**Source_Address**	
	Source_Address	Source memory block of 512bytes.
	The arguments can be a variable, array element, expression or constant	
Returns	Returns TRUE if media response was TRUE.	
Description	Writes 512 bytes (256 words) from a source memory block (eg wrblock[256]) into the uSD card. After the write the Sect pointer is automatically incremented by 1. Returns TRUE if uSD response was TRUE	
Example	`var wrblock[256];` `func main()` `prepare_block();` `media_SetSector(0,10)` `if (media_WrSector(wrblock));` `Print("Data transferred");` `endif` `:` `:` This example sets a 512 bytes block and transfers data to the address pointed to by media_SetSector command.	

2.8.6 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.

2.8.7 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”

2.8.8 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:** Writing bytes or words to a media sector must start from the beginning of the sector. All writes will be incremental until the **media_Flush()** function is executed, or the sector address rolls over to the next sector. When **media_Flush()** is called, 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.	

2.8.9 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	**Note: Writing bytes or words to a media sector must start from the beginning of the sector. All writes will be incremental until the media_Flush() function is executed, or the sector address rolls over to the next sector. When media_Flush() is called, 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.
```

2.8.10 media_Flush()

Syntax	**media_Flush();**
Arguments	**none**
Returns	returns 0 if Failed returns non-zero if OK
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.

2.8.11 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 may not be displayed correctly.	
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.	

2.8.12 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 may not 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.	
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.	

2.8.13 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 it may not 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 media_VideoFrame(..) will now show error box for out of range video frames. Also, if frame is set to -1, just a rectangle will be drawn in background colour to blank an image. It applies to PmmC R29 or above.	
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 Picaso 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 Picaso are the M25Pxx range of devices which are 512Kbit to 32Mbit (2M x 8) Serial Flash Memory.	
Example	`var sig, id;` `spi_Init(SPI_SLOW, RXMODE_0, CKMODE_0);` `//...` `sig := flash_SIG();` `id := flash_ID();` `print("Flash Signature:", [HEX4] sig, "\n") ;` `print("Flash (JEDEC)ID:", [HEX4] id, "\n") ;`	

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 Picaso are the M25Pxx range of devices which are 512Kbit to 32Mbit (2M x 8) Serial Flash Memory.	
Example	See the example in section **2.9.1 flash_SIG()**.	

2.9.3 flash_BulkErase()

Syntax	**flash_BulkErase();**
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 Picaso 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 Picaso 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.

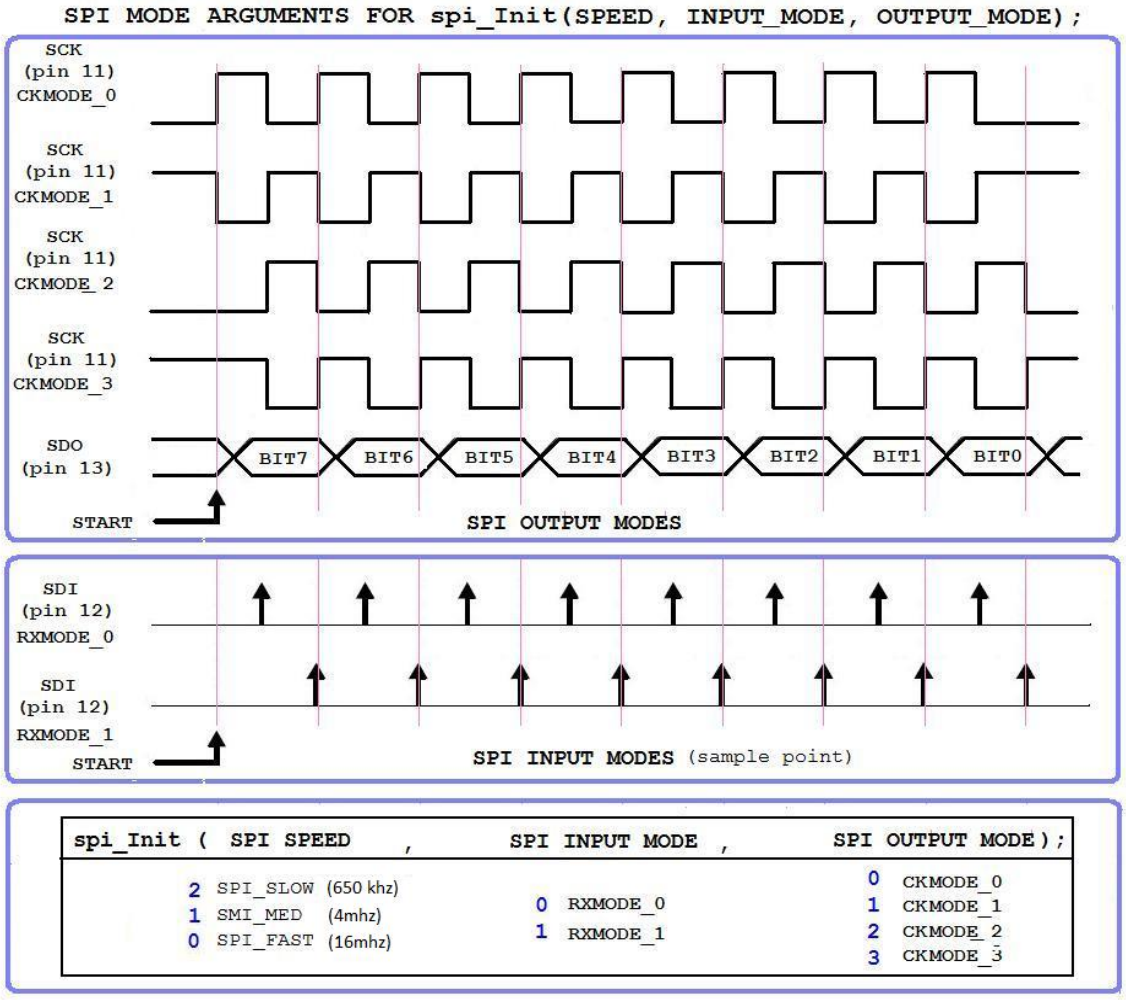
Summary of Functions in this section:

- 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 Picaso SPI port to communicate with SPI devices. See the example in section spi_Read().	

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.	
Example	`var result;` `spi_Init(SPI_SLOW, RXMODE_0, CKMODE_0);` `print("Hello World\n") ; // replace with your code` `//...` `spi_Write(0x40); // x_Accel Request` `result := spi_Read();` `print("result: ", result);`	

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.	
Example	See the example in section spi_Read().	

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:

- setbaud(rate)
- com_SetBaud(comport, baudrate/10)
- serin() or serin1()
- serout(char) or serout1(char)
- com_Init(buffer, buffsize, qualifier) or com_Init(buffer, buffsize, qualifier)
- com_Reset() or com1_Reset()
- com_Count() or com1_Count()
- com_Full() or com1_Full()
- com_Error() or com1_Error()
- com_Sync() or com1_Sync()
- com_TXbuffer(buf, bufsize,pin) or com1_TXbuffer(buf, bufsize,pin)
- com_TXbufferHold(state)
- com_TXcount() or com1_TXcount()
- com_TXemptyEvent(function) or com1_TXemptyEvent(function)

2.11.1 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 for COM0 is 115,200 bits per second or 115,200 baud. The default Baud Rate for COM1 is 9600 bits per second or 9600 baud. There are pre-defined baud rate constants for most common baud rates:	

Rate / Predefined Constant	Error %	Actual Baud Rate
BAUD_110	0.00%	110
BAUD_300	0.00%	300
BAUD_600	0.01%	600
BAUD_1200	0.03%	1200
BAUD_2400	0.07%	2402
BAUD_4800	0.16%	4808
BAUD_9600	0.33%	9632
BAUD_14400	0.16%	14423
BAUD_19200	0.33%	19264
BAUD_31250	0.00%	31250
MIDI	0.00%	31250
BAUD_38400	0.33%	38527
BAUD_56000	0.45%	56250
BAUD_57600	1.73%	58594
BAUD_115200	1.73%	117188
BAUD_128000	4.63%	133929
BAUD_256000	9.86%	281250
BAUD_300000	4.17%	312500
BAUD_375000	7.14%	401786
BAUD_500000	12.50%	562500
BAUD_600000	17.19%	703125

Example

```
setbaud(BAUD_19200);      // To set Com0 to 19200 BAUD rate.
```

2.11.2 com_SetBaud(comport, baudrate/10)

Syntax	**com_SetBaud("comport", "baudrate/10");**	
Arguments	**comport, baudrate/10**	
	comport	Com port, COM0 or COM1.
	baudrate/10	Specifies the baud rate.
	The arguments can be a variable, array element, expression or constant	
Returns	**Status**	
	Status	**Returns True if BAUD rate was acceptable.**
Description	Use this function to set the required baud rate for the required Com port. **Note:** Baud Rates are not always precise, and an approximate error can be seen from the setbaud() functions table on the previous page.	
Example	`stat := com_SetBaud(COM1, 960);  // To set Com1 to 9600 BAUD rate.` `if (stat)` `  Print("Com1 set to 9600 BAUD");` `endif`	

2.11.3 serin()

Syntax	**serin(); or serin1();**	
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	**serin():** Receives a character from the Serial Port COM0. **serin1():** Receives a character from the Serial Port COM1. The transmission format is: **No Parity, 1 Stop Bit, 8 Data Bits** (N,8,1). The default Baud Rate for COM0 is 115,200 bits per second or 115,200 baud. The default Baud Rate for COM1 is 9600 bits per second or 9600 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`	

2.11.4 serout(char)

Syntax	**serout(char); or serout(1char);**	
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	**serout()**: Transmits a single byte from the Serial Port COM1. **serout1()**: Transmits a single byte from the Serial Port COM1. The transmission format is: **No Parity, 1 Stop Bit, 8 Data Bits** (N,8,1). The default Baud Rate for COM0 is 115,200 bits per second or 115,200 baud. The default Baud Rate for COM1 is 9600 bits per second or 9600 baud. The baud rate can be changed under program control by using the **setbaud(...)** function. **serout()** normally blocks until the character can be transmitted, to enable serout to be non-blocking see **com_TXbuffer()**	
Example	`serout('\n');  \\Send a linefeed to COM0.`	

2.11.5 com_Init(buffer, bufsize, qualifier)

Syntax	**com_Init(buffer, bufsize, qualifier); or com1_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. **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.
Example	`com_Init(combuf, 20, 0 );` `// set up a comms ring buffer, maximum 12 characters before overflow`

2.11.6 com_Reset()

Syntax	**com_Reset(); or com1_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(); or com1_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(); or com1_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(); or com1_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(); or com1_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_TXbuffer(buf, bufsize,pin)

Syntax	**com_TXbuffer(buf, bufsize, pin); or com1_TXbuffer(buf, bufsize, pin);**	
Arguments	**buf, bufsize, pin**	
	buf	Specifies the address of a buffer used for the buffering service.
	bufsize	Specifies the byte size of the user array provided for the buffer (each array element holds 2 bytes).
	pin	Specifies the turnaround pin. If not required, just set "pin" to zero.
	The arguments can be a variable, array element, expression or constant	
Returns	**None**	
Description	Initialise a serial buffer for the COM0 or COM1 output. The program must declare a var array as a circular buffer. When a TX buffer is declared for comms, the transmission of characters becomes non-blocking. The only time blocking will occur is if the buffer has insufficient space to accept the next character, in which case the function will wait for buffer space to become available. If the TX buffer is no longer required, just set the buffer pointer to zero, the size in this case doesn't matter and is ignored. The function can resize or reallocated to another buffer at any time. The buffer is flushed before any changes are made. "pin" designates an IO pin to control a bi-directional control device for half duplex mode. "pin" will go HI at the start of a transmission, and will return low after the final byte is transmitted. Once the buffer has been initialised you just continue to use **serout()** in the usual way, no other programming changes are required	
Example	`com_TXbuffer(mybuf, 1024, IO1_PIN);  // set the TX buffer` `com_TXbuffer(0, 0, 0);  // revert to non buffered service`	

2.11.12 com_TXbufferHold(state)

Syntax	**com_TXbufferHold(state); or com1_TXbufferHold(state);**	
Arguments	**state**	
	state	Specifies the state of the buffer used for the buffering service.
	The arguments can be a variable, array element, expression or constant	
Returns	**count**	
	count	Returns buffer count when called with argument of 1, for example **com_TXbufferHold(ON)** Returns 0 when argument is zero, eg **com_TXbufferHold(OFF)**
Description	This function is used in conjunction with **com_TXbuffer(...)**; . It is often necessary to hold off sending serial characters until a complete frame or packet has been built in the output buffer. **com_TXbufferHold(ON)** is used for this, to stop the buffer being sent while it is being loaded. Normally, when using buffered comms, the transmit process will begin immediately. This is fine unless you are trying to assemble a packet. To build a packet and send it later, issue a **com_TXbufferHold(ON)**;, build the packet, when packet is ready, issuing **com_TXbufferHold(OFF)**; will release the buffer to the com port. Also, if using com_TXemptyEvent, erroneous empty events will occur as the transmit buffer is constantly trying to empty while you are busy trying to fill it. Also refer to the pin control for com_TXbuffer(...) function. Note: If you fill the buffer whilst it is held comms error 4 will be set and the data written will be lost.	
Example	Refer to the **com_TXemptyEvent(functionAddress)** example.	

2.11.13 com_TXcount()

Syntax	**com_TXcount(); or com1_TXcount();**	
Arguments	**None**	
Returns	**count**	
	count	Returns count of characters
Description	Return count of characters remaining in COM0 or COM1 transmit buffer that was previously allocated with com_TXbuffer(...); or com1_TXbuffer(...);	
Example	`arg := com1_TXCount(); //return count of characters in COM1 TX buffer`	

2.11.14 com_TXemptyEvent(function)

Syntax	**com_TXemptyEvent(functionAddress); or com1_TXemptyEvent(functionAddress);**	
Arguments	**functionAddress**	
	functionAddress	Address of the event Function to be queued when COM0 TX buffer empty
Returns	**Address**	
	Address	Returns any previous event function address, or zero if there was no previous function.
Description	If a comms TX buffer that was previously allocated with **com_TXbuffer(...);** or **com1_TXbuffer(...);**, this function can be used to set up a function to be called when the COM0 or COM1 TX buffer is empty. This is useful for either reloading the TX buffer, setting or clearing a pin to change the direction of eg a RS485 line driver, or any other form of traffic control. The event function must not have any parameters. To disable the event, simply call **com_TXemptyEvent(0)** or **com1_TXemptyEvent(0)**. **com_TXbuffer(...);** or **com1_TXbuffer(...);** also resets any active event.	
Example		

```
#platform "uLCD-32PT_GFX2"

/************************************************
* Description: buffered TX service
* Use Workshop terminal at 9600 baud to see result
* Example of Buffered TX service vs Non buffered
* Also explains the use of COMMS events
*
* NB Program must be written to flash so
* the Workshop Terminal can be used.
*
************************************************/

var combuf[220];  // buffer for up to 440 bytes

// run a timer event while we are doing comms
func T7Service()
   var private colour := 0xF800;
   colour ^= 0xF800;
   gfx_RectangleFilled(50,200,80,220,colour);
   sys_SetTimer(TIMER7, 200);
endfunc

// event to capture the buffer empty event
func bufEmpty()
   com_TXbuffer(0, 0, IO1_PIN);   // done with the buffer, release it
   print("\n\nHELLO WORLD, I'M EMPTY ",com_TXcount(),"\n");
endfunc
func main()
    var n, r, D, fh;

    sys_SetTimerEvent(TIMER7,T7Service);      // run a timer event
    sys_SetTimer(TIMER7, 150);
    com_TXemptyEvent(bufEmpty); // set to capture  buffer empty event

    setbaud(BAUD_9600);

    txt_Set(TEXT_OPACITY, OPAQUE);

repeat
```

```
    gfx_Cls();

    txt_MoveCursor(3,1);        // reset cursor to line 3, column 2
    print("Send 440 chars non-buffered\n");
    pokeW(SYSTEM_TIMER_LO, 0);  // reset timer

    // note that 440 chars at 9600 baud takes approx 453msec
    for(n:=0; n<10; n++)
        to(COM0); putstr("The quick brown fox jumps over the lazy dog\n");
// 44 chars
    next

    print("took ",peekW(SYSTEM_TIMER_LO),"Msec\n\n");
    // time spent blocking is only approx 1msec

    com_TXbuffer(combuf, 440,IO1_PIN);// set up the TX buffer
    com_TXbufferHold(ON);             // hold the TX buffer til ready


    // note that here the time is only approx 1msec overhead due to buffering.
    print("Send 440 chars buffered\n");
    pokeW(SYSTEM_TIMER_LO, 0);        // reset timer

    for(n:=0; n<10; n++)
        to(COM0); putstr("THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG\n");
// 44 chars
    next

    print("took ",peekW(SYSTEM_TIMER_LO),"Msec\n\n");
    // time spent blocking is only approx 1msec

    // demonstrate how to modify a prepared comms buffer that is
    still being held
    to(combuf); print("MY CONTENTS HAVE BEEN CHANGED");
    to(combuf+50); print("*** AND CHANGED HERE TOO ***");
    combuf[218] := 'CA';    // the last 'DOG' changed here
    combuf[219] := 'T\n';    // the last 'DOG' changed here

    // now we are ready to send to buffer
    n := com_TXbufferHold(OFF);   // release TX buffer
    print("TXBuffer is holding ", n, " chars\n");
    // show how many characters were in the buffer


    // watch the buffer empty
    repeat
        print("TX count = ", [DEC5ZB] n := com_TXcount(),"\r");   // watch
the count as the buffer empties
    until(!n);

    print("\n\nTX Empty");

    com_TXbuffer(0, 0, IO1_PIN); // done with the buffer, release it


    sys_SetTimer(TIMER0, 3000);  // pause for 3 seconds, non blocking
    while(peekW(TMR0));

forever // do it forever
//com_TXbuffer(0, 0, 0); // if done with the pin, must release it
endfunc
```

2.12. I2C BUS Master Functions

Summary of Functions in this section:

- func I2C_Open(Speed)
- func I2C_Close()
- func I2C_Start()
- func I2C_Stop()
- func I2C_Restart()
- func I2C_Read()
- func I2C_Write(byte)
- func I2C_Ack()
- func I2C_Nack()
- func I2C_AckStatus()
- func I2C_AckPoll(control)
- func I2C_Idle()
- func I2C_Gets(buffer, size)
- func I2C_Getn(buffer, size)
- func I2C_Puts(buffer)
- func I2C_Putn(buffer,count)

2.12.1 I2C_Open(Speed)

Syntax	**I2C_Open(Speed);**	
Arguments	**Speed**	
	Speed	Specifies the I2C bus speed Speed can be I2C_SLOW, I2C_MED, I2C_FAST (100khz, 400khz, 1mhz)
	The arguments can be a variable, array element, expression or constant	
Returns	**None**	
Description	Calling this function configures the I2C module and initialises it to be ready for service. The I2C clock speed is specified by the **speed** parameter. Three I2C Speed settings are available to suit various requirements. **Constant Speed** I2C_SLOW 100khz I2C_MED 400khz I2C_FAST 1mhz	
Example	`I2C_Open(I2C_MED); // Open the I2C port in 400KHz mode.`	

2.12.2 I2C_Close()

Syntax	**I2C_Close();**
Arguments	**None**
Returns	**None**
Description	Calling this function closes the I2C port and disables the I2C hardware
Example	`I2C_Close(); // Close I2C port and Disable the hardware`

2.12.3 I2C_Start

Syntax	**I2C_Start();**
Arguments	**None**
Returns	**None**
Description	Calling this function sends an I2C start condition. The hardware first pulls the SDA (data) line low, and next pulls the SCL (clock) line low. SCL ‾‾‾‾‾‾‾_____ SDA ‾‾‾‾_____
Example	`I2C_Start(); //Send an I2C start condition.`

2.12.4 I2C_Stop

Syntax	**I2C_Stop();**
Arguments	**None**
Returns	**None**
Description	Calling this function sends an I2C stop condition. The hardware first releases the SCL to high state, and then releases the SDA line high. SCL ____/ ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾ SDA _____/ ‾‾‾‾‾‾‾‾‾‾‾‾‾‾
Example	`I2C_stop(); //`

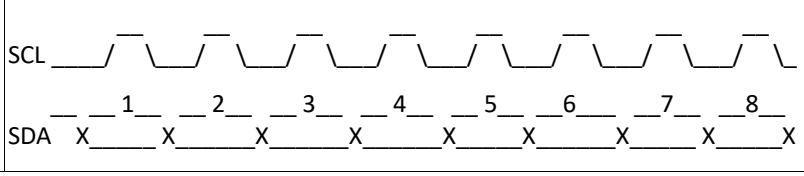
2.12.5 I2C_Restart()

Syntax	**I2C_Restart();**
Arguments	**None**
Returns	**None**
Description	Calling this function generates a restart condition.
Example	`I2C_Restart() ; //Generates an I2C restart condition`

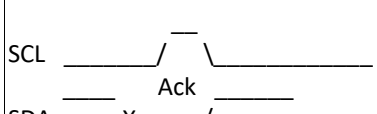
2.12.6 I2C_Read

Syntax	**I2C_Read();**	
Arguments	**None**	
Returns	**Byte**	
	Byte	**Byte from the I2C Bus in the lower 8 bits.**
Description	Calling this function reads a single byte from the I2C bus. Note: Data can only change when the clock is low. SCL ____/ ___/ ___/ ___/ ___/ ___/ ___/ ___/ ___/ _ __ __ 1__ __2__ __3__ __4__ __5__ __6___ __7__ __8__ SDA X_____X_____X_____X_____X_____X_____X_____X_____X	
Example	`c := I2C_Read() ; //Read a single byte from the I2C Bus.`	

2.12.7 I2C_Write(byte)

Syntax	**I2C_Write(byte);**	
Arguments	**byte**	
	byte	The byte to be written to the I2C Bus.
	The arguments can be a variable, array element, expression or constant	
Returns	**Status**	
	Status	**Returns 0** if False/Fail **Returns 1** if Success/OK **Returns 2** if NAK from device (or device does not exist)
Description	Calling this function sends a single byte to the I2C bus  	
Example	`Status := I2C_Write(bytevalue);// Send a single byte to the I2C`	

2.12.8 I2C_Ack

Syntax	**I2C_Ack();**
Arguments	**None**
Returns	**None**
Description	Calling this function sends an I2C acknowledge condition. The hardware first pulls the SDA line low, and next releases SCL high followed by pulling SCL low again thus generating a clock pulse, SDA is then released high. NB:- Data can only change when the clock is low. 
Example	`I2C_Ack(); // Send I2C Acknowledge condition`

2.12.9 I2C_Nack()

Syntax	**I2C_Nack();**
Arguments	**None**
Returns	**None**
Description	Calling this function sends an I2C negative acknowledge condition. The hardware first release the SDA line high, and next releases SCL HI followed by pulling SCL low thus generating a clock pulse. NB:- Data can only change when the clock is low. SCL _____/ _____ SDA ____ X ____ Nack
Example	`I2C_Nack(); //Send an I2C Negative acknowledge condition`

2.12.10 I2C_AckStatus

Syntax	**I2C_AckStatus();**	
Arguments	**None**	
Returns	**Status**	
	Status	**Device Ack status**
Description	Call this function to get the ACK status from the slave device The state of SDA is returned. NB:- returns the state of SDA after the last clock pulse __ Previous Clock Pulse SCL X _____ ____ SDA ____ X ____ Ack Status	
Example	`r := I2C_AckStatus();// returns the Ack Status.`	

2.12.11 I2C_AckPoll(control)

Syntax	**I2C_AckPoll(control);**	
Arguments	**control**	
	control	The control word to be written to the device.
	The arguments can be a variable, array element, expression or constant	
Returns	**Status**	
	Status	**Device Ack Status**
Description	Call this function to wait for a device to return an ACK during ACK polling The SDA is monitored for an Ack. NB:- returns the state of SDA after the last clock pulse __ Previous Clock Pulse SCL X _____ ____ SDA ____ X ____ Ack Status	
Example	`r := I2C_AckPoll(0xA0);//send the control byte the wait for a device` `//to return poll the device until an ACK` `//is received.`	

2.12.12 I2C_Idle()

Syntax	**I2C_Idle();**	
Arguments	**None**	
Returns	**Status**	
	Status	**Device Ack Status**
Description	Call this function to wait until the I2C bus is inactive. NB:- wait for the bus to become idle. SCL X___X/ SDA X___X/	
Example	`r := I2C_Idle(); //Wait until the I2C Bus is inactive.`	

2.12.13 I2C_Gets(buffer, size)

Syntax	**I2C_Gets(buffer, size);**	
Arguments	**buffer, size**	
	buffer	Storage for the string being read from the device.
	Size	Maximum size of the string to be read
Returns	**Count**	
	Count	**Returns the count of bytes actually read.**
Description	Reads up to **size** characters into **buffer** from an ascii string stored in a device. Reads up to the ASCII NULL terminator and includes the terminator.	
Example	`c := I2C_Gets(buf, size); //read a string from the I2C Bus to buffer` `//up to size characters.`	

2.12.14 I2C_Getn

Syntax	**I2C_Getn(buffer, count);**	
Arguments	**buffer, count**	
	buffer	Storage for the bytes being read from the device.
	count	Number of bytes to be read
	The arguments can be a variable, array element, expression or constant	
Returns	**Status**	
	Status	**Returns True if block read ok else returns False.**
Description	Reads count bytes in to buffer and returns True if function succeeds	
Example	`I2C_Getn(buffer, count); //read I2C count bytes from the I2C Bus to //the buffer`	

2.12.15 I2C_Puts(buffer)

Syntax	**I2C_Puts(buffer);**	
Arguments	**buffer**	
	buffer	Storage for the string being written to the device.
	The arguments can be a variable, array element, expression or constant	
Returns	**Count**	
	Count	**Returns the count of bytes actually written.**
Description	Writes an ASII string from buffer to a device. The ASCII NULL terminator is also written.	
Example	`c := I2C_Puts(mybuf); //write an ASCII string from buffer to the I2C //bus`	

2.12.16 I2C_Putn

Syntax	**I2C_Putn(buffer, count);**	
Arguments	**buffer, count**	
	buffer	Storage for the bytes being written to the device.
	count	Number of bytes to be written
Returns	**Count**	
	Count	**Returns number of bytes written.**
Description	Writes count bytes from the buffer to the device, and returns count if function succeeds.	
Example	`b := I2C_Putn(mybuf, count); // write count bytes from the buffer to` `// the I2C bus.`	

2.13. Timer Functions

Summary of Functions in this section:

- sys_T()
- sys_T_HI()
- sys_SetTimer(timernum, value)
- sys_GetTimer(timernum)
- sys_SetTimerEvent("timernum","function")
- sys_EventQueue()
- sys_EventsPostpone()
- sys_EventsResume()
- sys_DeepSleep(units)
- sys_Sleep(units)
- iterator(offset)

2.13.1 sys_T()

Syntax	**sys_T();**	
Arguments	**None**	
Returns	**value**	
	value	**Returns the value of system timer. (LO Word)**
Description	Returns the current value of the rolling 32bit system timer (1mse) LO word.	
Example	`t := sys_T(); // .`	

2.13.2 sys_T_HI()

Syntax	**sys_T_HI();**	
Arguments	**None**	
Returns	**value**	
	value	**Returns the value of system timer. (HI Word)**
Description	Returns the current value of the rolling 32bit system timer (1mse) HI word.	
Example	`t := sys_T_HI(); //`	

2.13.3 sys_SetTimer(timernum, value)

Syntax	**sys_SetTimer(timernum, value);**	
Arguments	**timernum, value**	
	timernum	One of eight timers TIMER0 to TIMER7.
	value	Countdown period in milliseconds.
	The “value” can be a variable, array element, expression or constant	
Returns	**None**	
Description	Set a countdown on the selected timer or 'top-up' if required. There are 8 timers TIMER0 to TIMER7 which stop at the count of 0. Maximum timeout period is 65535 milliseconds or 65.535 seconds. A timer can be read with the sys_GetTimer("timernum") function.	
Example	`sys_SetTimer(TIMER5, 3600); //Set Timer5 for 3.6 seconds`	

2.13.4 sys_GetTimer(timernum)

Syntax	**sys_GetTimer(timernum);**	
Arguments	**timernum**	
	timernum	One of eight timers TIMER0 to TIMER7.
Returns	**Value**	
	Value	**Returns 0 if timer has expired, or the current countdown value.**
Description	Returns 0 if timer has expired, or the current countdown value. There are 8 timers **TIMER0** to **TIMER7** which stop at the count of 0. Maximum timeout period is 65, 535 milliseconds or 65.535 seconds. A timer can be set with the sys_SetTimer("timernum", "value") function.	
Example	`t := sys_GetTimer(TIMER2); //`	

2.13.5 sys_SetTimerEvent(timernum, function)

Syntax	**sys_SetTimerEvent(timernum, function);**	
Arguments	**timernum, function**	
	timernum	One of eight timers TIMER0 to TIMER7.
	function	Event Function to be queued
Returns	**Address**	
	Address	**Returns any previous event function address, or zero if there was no previous function.**
Description	Set a function to be called for selected timer. When the timer reaches zero, the function is called. The called function must not have any parameters, and should not have a return value. This is necessary because the timer event is invoked asynchronously to the mainline program (i.e, it is not called in the normal way, so parameters and return values don't apply). **Note:** When a child process is run using the file_run or file_exec function, or if a file was loaded with file_Loadfunction and is executed, the loaded process gets its own code and memory space, therefore, any timer that reaches zero that has a timer event attached in the parent code space, will fail and cause a crash as an attempt is made to force the program counter to some wild place in the child process - There are 2 ways to overcome this problem. 1] If a child process will not be requiring the use of any timers or timer events, the parent program can simply use the eventsPostpone() function before calling or entering the child process. Once the parent program regains control, the eventsResume() function will allow any events in the queue to then be processed. The side effect of this method is that several events may bank up, and will execute immediately once the eventsResume() takes place. This however disallows a child process to use any timer events in the sub program so method 2 is preferable in this case. 2] The parent program can 'disconnect' the event(s) by setting it/them to zero prior to child process execution, or setting the associated timer to zero so the event wont fire. In either case, it is necessary to do the following:- while(sys_EventQueue()); to ensure the event queue is empty prior to calling the child process. Note also that if just the timer is set to zero, the child process cannot use this timer. If the timer was now set to a value and the old event still existed, when the timer reaches zero the 'bad' parent address event will fire causing a crash. The reverse situation also applies of course, the same level of respect is required if a child program needs to use any timer events. Method [1] (above) will not work as the events have been postponed, stopping the child process from using any timer events. If the child process did an eventsResume() in this case, everything would crash miserably. So the same applies, a child that uses any timer events must respect any timers that may be used by the parent, and a child must zero the sys_SetTimerEvent before returning to the parent. sys_SetTimerEvent(timernum, 0) disables the timer event.	
Example	`sys_SetTimerEvent(TIMER5, myfunc);`	

2.13.6 sys_EventQueue()

Syntax	**sys_EventQueue();**	
Arguments	**None**	
Returns	**Count**	
	Count	**Returns number of events .**
Description	returns the max number of events that were pending in the queue since the last call to this function. This can be used to assess event overhead burden, especially after or during a sys_EventsPostpone action..	
Example	`tasks := sys_EventQueue(); //`	

2.13.7 sys_EventsPostpone()

Syntax	**sys_EventsPostpone();**
Arguments	**None**
Returns	**None**
Description	Postpone any events until the sys_EventResume function is executed. The event queue will continue to queue events, but no action will take place until a sys_EventResume function is encountered. The queue will continue to receive up to 32 events before discarding any further events. This function is required to allow a sequence of instructions or functions to occur that would otherwise be corrupted by an event occurring during the sequence of instructions or functions. A good example of this is when you set a position to print, if there was no way of locking the current sequence, an event may occur which does a similar thing, and a contention would occur - printing to the wrong position. This function should be used wisely, if any action that is required would take considerable time, it is better to disable any conflicting event functions with a bypass flag, then restart the conflicting event by re-issuing a timer value.
Example	`sys_EventsPostpone();  // postpone the event queue`

2.13.8 sys_EventsResume()

Syntax	**sys_EventsResume();**
Arguments	**None**
Returns	**None**
Description	Resume any postponed events. The queue will try to execute any events that were incurred during the postponed period. Note that queued events are only checked for and executed at the the end of each 4DGL instruction.
Example	`sys_EventsResume();  // resume the event queue`

2.13.9 sys_DeepSleep(units)

Syntax	**sys_DeepSleep(units);**	
Arguments	**units**	
	units	Sleep timer units are approx 1 second. When in sleep mode, timing is controlled by an RC oscillator, therefore, timing is not totally accurate and should not be relied on for timing purposes
	The arguments can be a variable, array element, expression or constant	
Returns	**Status**	
	Status	**Remaining time units when touch screen is touched, else returns zero.**
Description	Put the display and processor into lowest power mode for a period of time. If "units" is zero, the display goes into sleep mode forever and needs power cycling to re-initialize. If "units" is 1 to 65535, the display will sleep for that period of time, or will be woken when touch screen is touched. The function returns the count of "units" that are remaining when the screen was touced. When returning from deep sleep mode, the processor is restored from low power mode, the display should be reinitialised with disp_Init(). New in v3.8 PmmC	
Example	`sys_DeepSleep(60); // Sleep for 1 minute.`	

2.13.10 sys_Sleep(units)

Syntax	**sys_Sleep(units);**	
Arguments	**units**	
	units	Sleep timer units are approx 1 second. When in sleep mode, timing is controlled by an RC oscillator, therefore, timing is not totally accurate and should not be relied on for timing purposes
	The arguments can be a variable, array element, expression or constant	
Returns	**Status**	
	Status	**Remaining time units when touch screen is touched, else returns zero.**
Description	Put the display and processor into low power mode for a period of time. If "units" is zero, the display goes into sleep mode forever and needs power cycling to re-initialize. If "units" is 1 to 65535, the display will sleep for that period of time, or will be woken when touch screen is touched. The function returns the count of "units" that are remaining when the screen was touced. When returning from sleep mode, the display and processor are restored from low power mode. **Note**: Sys_Sleep() was found to have an issue in PmmC's prior to R33, the units value was not always near 1 second. This has been corrected in PmmC R33.	
Example	`sys_Sleep(60); // Sleep for 1 minute.`	

2.13.11 iterator(offset)

Syntax	**iterator_(offset);**	
Arguments	**offset**	
	offset	Offset size for the next ++ or - - command
	The arguments can be a variable, array element, expression or constant	
Returns	**None**	
Description	Sets the iterator size for the next postinc, postdec, preinc or predec by a specified value. The offset will return to 1 after the next operation.	
Example	`t := iterator(10); //`	

2.14. FAT16 File Functions

Summary of Functions in this section:

- file_Error()
- file_Count(filename)
- file_Dir(filename)
- file_FindFirst(fname)
- file_FindNext()
- file_Exists(fname)
- file_Open(fname, mode)
- file_Close(handle)
- file_Read(destination, size, handle)
- file_Seek(handle, HiWord, LoWord)
- file_Index(handle, Hisize, Losize, recordnum)
- file_Tell(handle, &HiWord, &LoWord)
- file_Write(Source, size, handle)
- file_Size(handle, &HiWord, &LoWord)
- file_Image(x, y, handle)
- file_ScreenCapture(x, y, width, height, handle)
- file_PutC(char, handle)
- file_GetC(handle)
- file_PutW(word, handle)
- file_GetW(handle)
- file_PutS(source, handle)
- file_GetS(*String, size, handle)
- file_Erase(fname)
- file_Rewind(handle)
- file_LoadFunction(fname.4XE)
- file_Run(fname..4XE, arglistptr)
- file_Exec(fname..4XE, arglistptr)
- file_LoadImageControl(fname1, fname2, mode)
- file_Mount()
- file_Unmount()
- file_PlayWAV

2.14.1 file_Error()

Syntax	**file_Error();**		
Arguments	**None.**		
Returns	**Error Code**		
	ERROR CODE	**ERROR NUMBER**	**ERROR DESCRIPTION**
	FE_OK	**0**	**IDE Function Succeeded**
	FE_IDE_ERROR	**1**	**IDE command execution error**
	FE_NOT_PRESENT	**2**	**CARD not present**
	FE_PARTITION_TYPE	**3**	**WRONG partition type, not FAT16**
	FE_INVALID_MBR	**4**	**MBR sector invalid signature**
	FE_INVALID_BR	**5**	**Boot Record invalid signature**
	FE_MEDIA_NOT_MNTD	**6**	**Media not mounted**
	FE_FILE_NOT_FOUND	**7**	**File not found in open for read**
	FE_INVALID_FILE	**8**	**File not open**
	FE_FAT_EOF	**9**	**Fat attempt to read beyond EOF**
	FE_EOF	**10**	**Reached the end of file**
	FE_INVALID_CLUSTER	**11**	**Invalid cluster value > maxcls**
	FE_DIR_FULL	**12**	**All root dir entry are taken**
	FE_MEDIA_FULL	**13**	**All clusters in partition are taken**
	FE_FILE_OVERWRITE	**14**	**A file with same name exist already**
	FE_CANNOT_INIT	**15**	**Cannot init the CARD**
	FE_CANNOT_READ_MBR	**16**	**Cannot read the MBR**
	FE_MALLOC_FAILED	**17**	**Malloc could not allocate the FILE struct**
	FE_INVALID_MODE	**18**	**Mode was not r.w.**
	FE_FIND_ERROR	**19**	**Failure during FILE search**
	FE_INVALID_FNAME	**20**	**Invalid Filename**
	FE_INVALID_MEDIA	**21**	**bad media**
	FE_SECTOR_READ_FAIL	**22**	**Sector Read fail**
	FE_SECTOR_WRITE_FAIL	**23**	**Sector write fail**
Description	Returns the most recent error code or 0 if there were no errors.		
Example	`e := file_Error(); // .`		

2.14.2 file_Count(filename)

Syntax	**file_Count(filename);**	
Arguments	**filename**	
	filename	Name of the file(s) for the search (passed as a string)
Returns	**Count**	
	Count	**Number of files that match the criteria.**
Description	Returns number of files found that match the criteria. The wild card character '*****'matches up with any combination of allowable characters and '**?**' matches up with any single allowable character.	
Example	`count := file_Count("*.4XE"); //Returns number of files with ".4XE".`	

2.14.3 file_Dir(filename)

Syntax	**file_Dir(filename);**	
Arguments	**filename**	
	filename	Name of the file(s) for the search (passed as a string)
Returns	**Count**	
	Count	**Number of files found that match the criteria.**
Description	Streams a string of file names that agree with the search key. Returns number of files found that match the criteria. The wild card character '*****' matches up with any combination of allowable characters and '**?**' matches up with any single allowable character.	
Example	`count := file_Dir("*.4XE"); //Returns number of files with ".4XE".`	

2.14.4 file_FindFirst(fname)

Syntax	**file_FindFirst(fname);**	
Arguments	**fname**	
	fname	Name of the file(s) for the search (passed as a string)
Returns	**Status**	
	Status	**1: If at least one file exists that satisfies the criteria.** **0: If no file satisfies the criteria.**
Description	Returns true if at least 1 file exists that satisfies the file argument. Wildcards are usually used so if file_FindFirst returns true, further tests can be made using file_FindNext(); to find all the files that match the wildcard class. Note that the stream behaviour is the same as file_Dir.	
Example	`if (file_FindFirst("*.4XE"))` `  print("File Found") ; // .` `endif`	

2.14.5 file_FindNext()

Syntax	**file_FindNext();**	
Arguments	**None**	
Returns	**Status**	
	Status	**1: If more files exist that satisfy the criteria set in the file_FindFirstt(fname).** **0: If no more files satisfy the criteria set in the file_FindFrist(fname)**
Description	Returns true if more file exists that satisfies the file argument that was given for file_FindFirst. Wildcards must be used for file_FindFirst, else this function willalways return zero as the only occurence will have already been found. Note that the stream behaviour is the same as file_Dir.	
Example	`while ((file_FindNext())` `  filecount++;` `wend`	

2.14.6 file_Exists(fname)

Syntax	**file_Exists(fname);**	
Arguments	**fname**	
	fname	Name of the file for the search (passed as a string)
Returns	**Status**	
	Status	**1: File found** **0: File not found**
Description	Tests for the existence of the file provided with the search key. Returns TRUE if found.	
Example	`if (file_Exists("fil1.4XE"))` `  print("File Found") ; // .` `endif`	

2.14.7 file_Open(fname, mode)

Syntax	**file_Open(fname, mode);**	
Arguments	**fname, mode**	
	fname	Name of the file to be opened (passed as a string)
	mode	FILE_READ: 'r' FILE_WRITE:'w' FILE_APPEND: 'a'
Returns	**handle**	
	handle	**Returns handle if file exists. Sets internal file error number accordingly (0 if no errors).**
Description	Returns handle if file exists. The file**"handle"** that is created is now used as reference for "filename" for further file functions such as file_Close(handle);, etc. For FILE_WRITE and FILE_APPEND modes ('w' and 'a') the file is created if it does not exist. If the file is opened for append and it already exists, the file pointer is set to the end of the file ready for appending, else the file pointer will be set to the start of the newly created file. If the file was opened successfully, the internal error number is set to 0 (ie:- no errors) and can be read with the file_Error(); function.. For FILE_READ mode ('r') the file must exist else a null handle (0) is returned and the 'file not found' error number is set which can be read with the file_Error(); function.. **Note:** If a file is opened for write mode 'w' , and the file already exists, the operation will fail. Unlike C and some other languages where the file will be erased ready for re-writing when opened for writing, 4DGL offers a simple level of protection that ensures that a file must be purposely erased before being re-written. **Note:** Beginning with the v4.0 PmmC a file opened with FILE_APPEND may be randomly read and or written. Also any altered file will have the Archive bit set in the directory entry.
Example	`handle := file_Open("myfile.txt", 'r');`	

2.14.8 file_Close(handle)

Syntax	**file_Close(handle);**	
Arguments	**handle**	
	handle	the file handle that was created by file_Open("fname") which is now used as reference (handle) for "fname" for further file functions such as in this function to close the file.
Returns	**Status**	
	Status	**1: File Closed.** **0: File not closed.**
Description	Returns TRUE if file closed, FALSE if not.	
Example	`res := file_Close(hndl);`	

2.14.9 file_Read(destination, size, handle)

Syntax	**file_Read(*destination, size, handle);**	
Arguments	**destination, size, handle**	
	destination	Destination memory buffer, this is a normal word aligned address
	size	Number of bytes to be read
	handle	The handle that references the file to be read.
Returns	**count**	
	count	**Returns the number of characters read.**
Description	Reads the number of bytes specified by "**size**" from the file referenced by "**handle**" into a destination memory buffer.	
Example	`res := file_Read(memblock, 20, hndl1);`	

2.14.10 file_Seek(handle, HiWord, LoWord)

Syntax	**file_Seek(handle, HiWord, LoWord);**	
Arguments	**handle, HiWord, LoWord**	
	handle	The handle that references the file
	HiWord	Contains the upper 16bits of the memory pointer into the file
	LoWord	Contains the lower 16bits of the memory pointer into the file
Returns	**Status**	
	Status	**Returns TRUE if ok, usually ignored**
Description	Places the file pointer at the required position in a file that has been opened in '**r**' (read) or '**a**' (append) mode. In append mode, file_Seek does not expand a filesize, instead, the file pointer (handle) is set to the end position of the file, eg:- assuming the file size is 10000 bytes, file_Seek(handle, 0, 0x1234); will set the file position to 0x00001234 (byte position 4660) for the file handle, so subsequent data may be read from that position onwards with file_GetC(...), file_GetW(...), file_GetS(...), or an image can be displayed with file_Image(...). Conversely, file_PutC(...), file_PutW(...) and file_PutS(...) can write to the file at the position. A **FE_EOF** (end of file error) will occur if you try to write or read past the end of the file.	
Example	`res := file_Seek(hSource, 0x0000, 0x1234) ;`	

2.14.11 file_Index(handle, Hisize, LoSize, recordnum)

Syntax	**file_Index(handle, Hisize, LoSize, recordnum);**	
Arguments	**handle, Hisize, LoSize, recordnum**	
	handle	The handle that references the file
	Hisize	Contains the upper 16bits of the size of the file records.
	LoSize	Contains the lower 16bits of the size of the file records.
	recordnum	The index of the required record
Returns	**Status**	
	Status	**Returns TRUE if ok, usually ignored**
Description	Places the file pointer at the position in a file that has been opened in '**r**' (read) or '**a**' (append) mode. In append mode, file_Index does not expand a filesize, instead, the file pointer (handle) is set to the end position of the file, eg:- assuming the record size is 100 bytes, file_Index(handle, 0, 100, 22); will set the file position to 2200 for the file handle, so subsequent data may be read from that position onwards with file_GetC(...), file_GetW(...), file_GetS(...), or an image can be displayed with file_Image(...). Conversely, file_PutC(...), file_PutW(...) and file_PutS(...) can write to the file at the position. A **FE_EOF** (end of file error) will occur if you try to write or read past the end of the file.	
Example	`res := file_Index(hSource, 0, 100, 22) ;`	

2.14.12 file_Tell(handle, &HiWord, &LoWord)

Syntax	**file_Tell(handle, &HiWord, &LoWord);**	
Arguments	**handle, &HiWord, &LoWord**	
	handle	The handle that references the file
	HiWord	Contains the upper 16bits of the memory pointer into the file
	LoWord	Contains the lower 16bits of the memory pointer into the file
Returns	**Status**	
	Status	**Returns TRUE if ok, usually ignored**
Description	Returns the current value of the file pointer.	
Example	`res := file_Tell(hSource, &HIptr, &LOptr) ;`	

2.14.13 file_Write(*source, size, handle)

Syntax	**file_Write(*source, size, handle);**	
Arguments	**source, size, handle**	
	source	Source memory buffer, this is a byte aligned string pointer
	size	Number of bytes to be written.
	handle	The handle that references the file to write.
Returns	**count**	
	count	**Returns the number of bytes written.**
Description	Writes the number of bytes specified by "**size**" from the source buffer into the file referenced by "**handle**".	
Example	`res := file_Write(memblock, 20, hndl1);`	

2.14.14 file_Size(handle, &HiWord, &LoWord)

Syntax	**file_Size(handle, &HiWord, &LoWord);**	
Arguments	**handle, HiWord, LoWord**	
	handle	The handle that references the file.
	HiWord	Contains the upper 16bits of the file size.
	LoWord	Contains the lower 16bits of the file size.
Returns	**Status**	
	Status	**Returns TRUE if ok, usually ignored.**
Description	Reads the 32 bit file size and stores it into 2 variables	
Example	`res := file_Size(hSource, &sizeHi, &sizeLo);`	

2.14.15 file_Image(x, y, handle)

Syntax	**file_Image(x, y, handle);**	
Arguments	**x, y, handle**	
	x	X-position of the image to be displayed
	y	Y-position of the image to be displayed
	handle	The handle that references the file containing the image(s)
Returns	Returns a copy of the file_Error() error code	
Description	Display an image from the file stream at screen location specified by x, y(top left corner). If there is more than 1 image in the file, it can be accessed with file_Seek(...).	
Example	`file_Image(x, y, handle) ;`	

2.14.16 file_ScreenCapture(x, y, width, height, handle)

Syntax	**file_ScreenCapture(x, y, width, height, handle);**	
Arguments	**x, y, width, height, handle**	
	x	X-position of the image to be captured
	y	Y-position of the image to be captured
	width	Width of the area to be captured.
	height	Height of the area to be captured.
	handle	The handle that references the file to store the image(s)
Returns	**Status**	
	Status	**Returns 0 if function successful.**
Description	Save an image of the screen shot to file at the current file position. The image can later be displayed with file_Image(...); The file may be opened in append mode to accumulate multiple images. Later, the images can be displayed with file_Seek(...). The image is saved from x, y (with respect to top left corner), and the capture area is determined by "width" and "height".	
Example		

```
file_Mount();
hFile := file_Open("test.img", 'a'); // open a file to save the image
file_ScreenCapture(20,20,100,100, hFile);// save an area
file_ScreenCapture(0,0,50,50, hFile);    // (save another area)
file_Close(hFile);                       // now close the file

// and to display the saved area(s)

hFile := file_Open("test.img", 'r');    // open the saved file
file_Image(20,180, hFile);              // display the image
file_Image(150,180, hFile);             // (display the next image)
file_Close(hFile);
file_Unmount();                         // finished with file system
```

2.14.17 file_PutC(char, handle)

Syntax	**file_PutC(char, handle);**	
Arguments	**char, handle**	
	char	Data byte about to be written.
	handle	The handle that references the file to be written to.
Returns	**BytesWritten**	
	BytesWritten	Returns the number of bytes written
Description	This function writes the byte specified by "char" to the file, at the position indicated by the associated file-position pointer and advances the pointer appropriately (incremented by 1). The file must be previously opened with 'w' (write) or 'a' (append) modes.	
Example	`file_PutC('A', hndl);`	

2.14.18 file_GetC(handle)

Syntax	**file_GetC(handle);**	
Arguments	**handle**	
	handle	The handle that references the file.
Returns	**byte**	
	byte	**Returns the data byte read from the file.**
Description	This function reads a byte from the file, at the position indicated by the associated file-position pointer and advances the pointer appropriately (incremented by 1). The file must be previously opened with 'r' (read) mode.	
Example	`mychar := file_GetC(hndl) ;`	

2.14.19 file_PutW(word, handle)

Syntax	**file_PutW(word, handle);**	
Arguments	**word, handle**	
	word	Data about to be written
	handle	The handle that references the file to be written to.
Returns	**BytesWritten**	
	BytesWritten	Returns the number of bytes written
Description	This function writes word sized (2 bytes) data specified by "**word**" to the file, at the position indicated by the associated file-position pointer and advances the pointer appropriately (incremented by 2). The file must be previously opened with 'w' (write) or 'a' (append) modes.	
Example	`file_PutW(0x1234, hndl);`	

2.14.20 file_GetW(handle)

Syntax	**file_GetW(handle);**	
Arguments	**handle**	
	handle	The handle that references the file.
Returns	**Word**	
	Word	**Returns word sized data read from the file.**
Description	This function reads a word (2 bytes) from the file, at the position indicated by the associated file-position pointer and advances the pointer appropriately (incremented by 2). The file must be previously opened with 'r' (read) mode.	
Example	`myword := file_GetW(hndl);`	

2.14.21 file_PutS(*source, handle)

Syntax	**file_PutS(*source, handle);**	
Arguments	**source, handle**	
	source	A pointer to the string to be written.
	handle	The handle that references the file to be written to.
Returns	**count**	
	count	**Returns the number of characters written (excluding the null terminator).**
Description	This function writes an **ASCIIZ** (null terminated) string from a buffer specified by "***source**" to the file, at the position indicated by the associated file-position pointer and advances the pointer appropriately. The file must be previously opened with 'w' (write) or 'a' (append) modes.	
Example	`file_PutS(mystring, hndl);`	

2.14.22 file_GetS(*string, size, handle)

Syntax	**file_GetS(*string, size, handle);**	
Arguments	**string, size, handle**	
	string	Destination buffer
	size	The maximum number of bytes to be read from the file.
	handle	The handle that references the file.
Returns	**Count**	
	Count	**Returns the number of characters read from file (excluding the null teminator)**
Description	This function reads a line of text to a buffer (specified by "***string**") from a file at the current file position indicated by the associated file-position pointer and advances the pointer appropriately. Characters are read until either a newline or an EOF is received or until the specified maximum "**size**" is reached. In all cases, the string is null terminated. The file must be previously opened with 'r' (read) mode.	
Example	`res := file_GetS(mystring, 80, hndl);`	

2.14.23 file_Erase(fname)

Syntax	**file_Erase(fname);**	
Arguments	**fname**	
	fname	Name of the file to be erased
Returns	**Status**	
	Status	**1: if successful** **0: if unsuccessful**
Description	This function erases a file on the disk. Note: If the function fails, the approprialte error number is set in file_Error() and will usually be error 19, "failure during FILE search".	
Example	`res := file_Erase("myfile.txt") ;`	

2.14.24 file_Rewind(handle)

Syntax	**file_Rewind(handle);**	
Arguments	**handle**	
	handle	The handle that references the file
Returns	**Status**	
	Status	**Returns TRUE if ok, usually ignored**
Description	Resets the file pointer to the beginning of a file that has been opened in '**r**' (read), '**w**', or '**a**' (append) mode.	
Example	`res := file_Rewind(hSource); ;`	

2.14.25 file_LoadFunction(fname.4XE)

Syntax	**file_LoadFunction(fname.4XE);**	
Arguments	**fname.4XE**	
	fname.4XE	name of the 4DGL application program that is about to be loaded into RAM.
Returns	**Pointer**	
	Pointer	**Returns a pointer to the memory allocation where the function has been loaded from file which can be then used as a function call.**
Description	Load a function or program from disk and return a function pointer to the allocation. The function can then be invoked just like any other function would be called via a function pointer. Parameters may be passed to it in a conventional way. The function may be discarded at any time when no longer required, thus freeing its memory resources. The loaded function can be discarded with mem_Free(..) Note that any pointer references passed to the child function may not include references to the parents DATA statements or any static string references. Any string or array information must be in the parents global or local memory space. The reason for this is that DATA statements and static strincs are contained in the parents CODE segment, and cannot be accessed by the child process. **Note:** PmmC Rev 31 and above has an added feature where a parent can access the child global Variables when using file_LoadFunction(fname.4XE).	
Example1	`var titlestring[20];` `var textstring[20];` `to(titlestring); putstr("My Window Title");` `to (textstring); putstr("My Special Message");` `popupWindow := file_LoadFunction("popupWindow1.4fn");` `if(!popupWindow)goto LoadFunctionFailed;//could not load the function` `//then elsewhere in your program` `res := popupWindow(MYMODE,titlestring,textstring);` `if(res == QUIT_APPLICATION) goto exitApp;` `//Later in your program, when popupWindow is no longer required` `//for the application` `res := mem_Free(popupWindow);` `if(!res) goto FreeFunctionFailed; //should never happen if memory not` `//corrupted`	
Example2	`var fncHandle;   //a var for a handle to sliders2.4dg` `var slidervals; //reference var to access global vars in sliders.4dg` `fncHandle := file_LoadFunction("sliders2.4xe"); // load the function` `slidervals := fncHandle&0x7FFF;    // note that memory allocations` `for transient programs are biased with 8000h which must be removed.` `slidervals++;  // note that all globals start at '1'` `slidervals[0] := 25;   // set sliders to initial positions` `slidervals[1] := 20;` `slidervals[2] := 30;` `slidervals[3] := 15;` `slidervals[4] := 35;` `slidervals[5] := 20;`	

```
slidervals[6] := 40;
slidervals[7] := 25;
slidervals[8] := 45;
slidervals[9] := 5;

r := fncHandle();     // activate the function

print("Return value = 0x", [HEX] r,"\n");

// print the  values, they may have changed
print("Slider 1  ", slidervals[0]," Slider 2  ", slidervals[1],"\n");
print("Slider 3  ", slidervals[2]," Slider 4  ", slidervals[3],"\n");
print("Slider 5  ", slidervals[4]," Slider 6  ", slidervals[5],"\n");
print("Slider 7  ", slidervals[6]," Slider 8  ", slidervals[7],"\n");
print("Slider 9  ", slidervals[8]," Slider 10 ", slidervals[9],"\n");

mem_Free(fncHandle);  // done with sliders, release its memory
```

Note: Refer to the complete sample code in Workshop4:
File Menu -> Samples -> Picaso Designer -> Picaso-Loadfunctiontest -> LOADFUNCTIONTEST.4DG

2.14.26 file_Run(fname.4XE, arglistptr)

Syntax	**file_Run(fname.4XE, arglistptr);**	
Arguments	**fname.4XE, arglistptr**	
	fname.4XE	name of the 4DGL child program to be loaded into RAM and executed.
	arglistptr	pointer to the list of arguments to pass to the new program.
Returns	**Value**	
	Value	**Returns the value from main in the called program.**
Description	Any memory allocations in the main FLASH program are released, however, the stack and globals are maintained. func 'main' in the called program accepts the arguments, if any. If arglistptr is 0, no arguments are passed, else arglistptr points to an array, the first element containing the number of additional elements in the array which contain the arguments. The disk does not need to be mounted, file_Run automatically mounts the drive.	
Example		

```
#inherit "4DGL_16bitColours.fnc"
#inherit "FONT4.fnt"

#constant MAXBUTTONS 30 // for now, maximum number of buttons we want
                    // (also sets maximum number of files we can exec)

#STACK 500
//stack must be large enough to be shared with called program
#MODE RUNFLASH
// This is a 'top down' main program and must be run from FLASH

//---------------------------------------------------------------------//
local global variables
//---------------------------------------------------------------------
// NB:- demo assigns all arrays to MAXBUTTONS.
// The arrays could be dynamically assigned to minimise memory usage.
// There is break even point between extra code and smallish arrays.
var keyval;       // 0 if no key pressed else 1-n
var filenames;    // pointer to byte array that holds the filenames

var buttontexts[MAXBUTTONS]; // pointers into the filenames array
//holds the filenames we use as button text

var vButtonState[MAXBUTTONS];
//button state flag( bit 0 = up:down state)
var vOldButtonState[MAXBUTTONS];
// OLD button state flags (bit 0 = up:down state)

// (we keep 2 copies so we can test for a state change and only redraw when
a state change occurs)

var touchX1[MAXBUTTONS];           // touch regions for the buttons
var touchY1[MAXBUTTONS];
var touchX2[MAXBUTTONS];
var touchY2[MAXBUTTONS];

var btnTextColor;                       // button text colour
var btnBtnColor;                        // button background colour
var buttoncount;                    // actual number of buttons created (set
by number of *.4XE files we find on drive)
```

```
var tempstr[20];                // general purpose string, 40 bytes

#DATA
 byte fred 1,2,3,4,5,6,7,8,9,10,11,12
#END

/*=====================================================================
Redraw the button matrix. Only draw buttons that have changed state.
The top lef corner of the button matrix is set with the xorg and yorg
parameters depending on the font and text string width, the button matrix
dynamically resizes.
Parameters:-
maxwidth    = rhs from xorg (in pixels) to cause wrap at rhs
maxwidth    = maximum matrix width (in pixel units)
buttoncount = number of buttons to display
font        = FONT1 to FONT4
xorg:yorg   = top left corner of button array
NB:- The  touch detect matrix array is updated when any button changes state.
When you need to draw the matrix for the first instance of the matrix, you
must
call with mode = 1 to instantiate the buttons.
call with mode = 0 for normal button action.
=====================================================================*/

func redraw(var bcount, var font, var xorg, var yorg, var maxwidth, var mode
)

    var xgap, ygap, n, x1, y1, x2, y2;

    xgap := 2;
    ygap := 2;
    x1 := xorg;
    y1 := yorg;

    // if first, set all the buttons to the up state
    if (mode)
        n := 0;
        repeat
            vButtonState[n]:=UP;
// set all the buttons to inverse state
            vOldButtonState[n]:=DOWN;
// so we guarantee they are all drawn in the 'up' state (not pressed)
        until(++n >= buttoncount);
    endif

// check all the button states, if a change occured, draw the new button
state and update the touch detect matrix array
    n := 0;
    repeat
        // if the button state has changed
        if ( vButtonState[n] != vOldButtonState[n])
            vOldButtonState[n] := vButtonState[n];

            // if we already have all the co-ordinates, use them
            if (!mode)
                x1 := touchX1[n];
                y1 := touchY1[n];
                x2 := touchX2[n];
                y2 := touchY2[n];
            endif

            // draw the button
            gfx_Button( vButtonState[n], x1, y1, btnBtnColor, btnTextColor,
font, 1, 1, buttontexts[n] );

           // update the touch screen regions only during first build
            if (mode)
                x2 := gfx_Get(RIGHT_POS);
```

```
                y2 := gfx_Get(BOTTOM_POS);

                touchX1[n] := x1;
                touchY1[n] := y1;
                touchX2[n] := x2;
                touchY2[n] := y2;

                // calculate next button position
                x1 := x2 + xgap;
                if (x1 >= xorg + maxwidth)
                    x1 := xorg;
                    y1 := y2 + ygap;
                endif
            endif

        endif
    until (++n >= buttoncount);
endfunc

//=====================================================================
// do something with the key data
// In this example, we reconstitute the button name to a file name
// by appending ".4XE" and then call the file_Run command to
// run an application.
//=====================================================================
func sendkey()
    var p;

    p := buttontexts[keyval-1];
    to(tempstr); str_Printf(&p, "%s.4XE");

    txt_Set(TEXT_OPACITY, OPAQUE);
    txt_Set(FONT_ID , FONT4);
    txt_MoveCursor(3, 0);

    print ("                  ");

    if(file_Exists(str_Ptr(tempstr)))
        touch_Set(TOUCH_DISABLE);          // disable the touch screen
        txt_Set(TEXT_COLOUR, ORANGE);
        print ("\rRUN: ", [STR] tempstr );// run the required program
        pause(500);
        gfx_Cls();
        file_Run(str_Ptr(tempstr),0);    // just run the prog, no args
     else
        txt_Set(TEXT_COLOUR, RED);
        print ("\rFAULT: ", [STR] tempstr );  // run required program
        pause(1000);
    endif

endfunc

//=====================================================================
// convert the touch co-ordinates to a key value
// returns 0 if no key down else return index 1..n of button
//=====================================================================
func readKeys(var x, var y)

    var n, x1, y1, x2, y2, r;

    n := 0;
    r := 0;

    while (n < buttoncount && !r)
        x1 := touchX1[n];
        y1 := touchY1[n];
        x2 := touchX2[n];
        y2 := touchY2[n];
```

```
        n++;
        if (x >= x1 && x < x2 && y >= y1 && y < y2) r := n;
    wend

    return r;
endfunc

//=====================================================================
func main()

    var k, n, state, x, y;
    var p, s, w, f;
redo:
    w := 140;
    f := FONT4;
    btnTextColor := BLACK;
    btnBtnColor := LIGHTGREY;

    gfx_Cls();
    gfx_Set(BEVEL_WIDTH, 2);

    txt_Set(FONT_ID, FONT3);
    print("Simple test for file_Run(...);\n");
    print("Memory available = ",mem_Heap(),"\n");

    if(!file_Mount())
        putstr("Disk not mounted");
        while(!file_Mount());
    else
        putstr("Disk mounted\n");
    endif

    buttoncount := file_Count("*.4xe");
// count all the executable files on the drive
    print("4XE File count = ",buttoncount,"\n");

    n := buttoncount;       // k holds entry count
    if (!n)
        print("No 4XE executables\n");
// critical error, nothing to run!
        repeat forever
    endif

    filenames := mem_AllocZ(n*13);
// allocate a buffer for the filenames
    if(!filenames)
        print("Out of memory\n");
// critical error, could not allocate buffer
        repeat forever
    endif

    to(filenames); file_Dir("*.4xe");
// load the filenames array

    p := str_Ptr(filenames);    // point to the string

//assign array of string pointers and truncate filename extensions
    n := 0;
    while ( n < buttoncount )
        buttontexts[n++] := p;    // save pointer to the string
        p:=str_Find ( &p , "." ); // find end of required string
        str_PutByte(p++,'\0');    // change '.' to \0
        p := p + 4;               // skip over "4XE\n"
    wend

    touch_Set(TOUCH_ENABLE);      // enable the touch screen

    redraw(buttoncount, f, 10, 80, w, 1);
```

```
// draw buttons for the first time

    // now just stay in a loop
    repeat
        state := touch_Get(TOUCH_STATUS);  // get touchscreen status
        x := touch_Get(TOUCH_GETX);
        y := touch_Get(TOUCH_GETY);

       if(state == TOUCH_PRESSED)           // if there's a press
            if (keyval := readKeys(x, y))
                vButtonState[keyval-1] := DOWN;
// put button in DOWN state
                redraw(buttoncount, f, 10, 80, w, 0);
// draw any button down states
            endif
        endif

        if(state == TOUCH_RELEASED)
// if there's a release
            if (keyval)
                vButtonState[keyval-1] := UP;
// restore the buttons UP state
                redraw(buttoncount, f, 10, 80, w, 0);
// draw any button up states
                sendkey();
// do something with the key data
                keyval := 0;
// because prog(main prog) gave up all its allocations for file_Exec,
// we have lost our file mount info and the directory list so we must
// re-establish these to be able to continue. A better approach to
// ensure total stability for the main program is to reset the system
                // with SystemReset()
                //==================================
                // systemReset() // restart the main program
                // or
                goto redo;      // re-mount disk, reload filenames
                //==================================

            endif
        endif

    forever

    // mem_Free(filenames);
   // no need to release buffer, this prog is in flash and never exits.....
    // file_Unmount();                          // ditto

endfunc
//=====================================================================
```

2.14.27 file_Exec(fname.4XE, arglistptr)

Syntax	**file_Exec(fname.4XE, arglistptr);**	
Arguments	**fname.4XE, arglistptr**	
	fname.4XE	name of the 4DGL child program to be loaded into RAM and executed.
	arglistptr	pointer to the list of arguments to pass to the new program or 0 if no arguments.
Returns	**Value**	
	Value	**Returns the value from main in the called program.**
Description	This function is similar to **file_Run,** however, the main program in FLASH retains all memory allocations (eg file buffers, memory allocated with mem_Alloc etc)	
Example	**Main Program:** `var args[4], l[50] ;` `func main()` `  var i ;` `  putstr("Mounting...\n");  // must mount uSD for file_Exec` `  if (!(file_Mount()))` `  while(!(file_Mount()))` `  putstr("Drive not mounted...");` `  pause(200);` `  gfx_Cls();` `  pause(200);` `  wend` `  endif` `  for (i := 0; i < sizeof(l); i++)  // init array that will be passed` `  l[i] := i ;` `  next` `  args[0] := 2 ;  // init arg count` `  args[1] := 1234 ;  // init arg 1, this cannot be changed` `  args[2] := l ;  // init arg 2 to address of l` `  print("main Program\n" ) ;` `  i := file_Exec("uSDProg.4fn", args) ;` `  print("Back in main program\n" ) ;` `  print("uSD Program returned ", i, "\n") ; // number from return statement` `  for (i := 0; i < sizeof(l); i++)  // find what changed in array` `  if (l[i] != i) print("l[", i, "] was changed to ", l[i], "\n" ) ;` `  next` `  print("Done") ;` `  repeat` `  forever` `endfunc` **Function on uSD:** `func main(var j, var *l)  // parameters appear in the normal way` `  // The * shows that l will be indexed. It` `  // simply stops the compiler issuing a 'notice'` `  txt_FGcolour(WHITE);` `  print("In file_Exec's Program\n") ;`	

```
    print("Parms=", j, " ", l, "(ptr to l)\n") ;  // can't change these
    print("Incrementing l[5] to ", ++l[5], "\n") ; // can change these
    print("Returning 188\n") ;                    // can return a value
    txt_FGcolour(LIME);
    return 188 ;
endfunc
```

2.14.28 file_LoadImageControl(fname1, fname2, mode)

Syntax	**file_LoadImageControl(fname1, fname2, mode);**	
Arguments	**fname1, fname2, mode**	
	fname1	the control list filename "*.dat". Created from Graphics Composer.
	fname2	the image filename "*.gci". Created from Graphics Composer.
	mode	mode 0 : It is assumed that there is a graphics file with the file extension "fname2.gci". In this case, the images have been stored in a FAT16 file concurrently, and the offsets that are derived from the "fname1.dat" file are saved in the image control so that the image control can open the file (*.gci) and use file_Seek(..) to get to the position of the image which can then automatically be displayed using file_Image(xpos, ypos, hSource). Mode 0 builds the image control quickly as it only scans the *.dat file for the file offsets and saves them in the relevant entries in the image control. The penalty is that images take longer to find when displayed due to file_Seek(..) overheads. mode 1 : It is assumed that there is a graphics file with the file extension "fname2.gci". In this case, the images have been stored in a FAT16 file concurrently, and the offset of the images are saved in the image control so that image file (*.gci) can be mapped to directly. The absolute cluster/sector is mapped so file seek does not need to be called internally. This means that there is no seek time penalty, however, the image list takes a lot longer to build, as all the seeking is done at control build time. Mode 2 : In this case, the images have been stored in a in a RAW partition of the uSD card, and the absolute address of the images are saved in the DAT file. This is the fastest operation of the image control as there is no seeking or other disk activity taking place.
Returns	**Status**	
	Status	**Returns a handle (pointer to the memory allocation) to the image control list that has been created.** **Returns NULL if function fails.**
Description	Reads a control file to create an image list. When an image control is loaded, an array is built in ram. It consists of a 6 word header with the following entries as defined by the constants: IMG_COUNT 0 IMG_ENTRYLEN 1 IMG_MODE 2 IMG_GCI_FILENAME 3 IMG_DAT_FILENAME 4 IMG_GCIFILE_HANDLE 5 No images are stored in FLASH or RAM, the image control holds the index values for the absolute	

storage positions on the uSD card for RAW mode, or the cluster/sector position for formatted FAT16 mode.

When an image control is no longer required, the memory can be released with:

mem_Free(MyImageControlHandle);

Example

```
#inherit "4DGL_16bitColours.fnc"

#constant OK   1
#constant FAIL 0

    var p;                              // buffer pointer
    var img;                            // handle for the image list
    var n, exit, r;

//---------------------------------------------------------------------
// return true if screen touched, also sets ok flag
func CheckTouchExit()
    return (exit := (touch_Get(TOUCH_STATUS) == TOUCH_PRESSED));     // if
there's a press, exit
endfunc
//---------------------------------------------------------------------

func main()

    gfx_Cls();
    txt_Set(FONT_ID, FONT2);
    txt_Set(TEXT_OPACITY, OPAQUE);

    touch_Set(TOUCH_ENABLE);                // enable the touch screen

    print("heap=", mem_Heap(), " bytes\n");  // show the heap size

    r := OK; // return value
    exit := 0;

    if (!file_Mount())
        print("File error ", file_Error());
        while(!CheckTouchExit());
// just hang if we didnt get the image list
        r := FAIL;
        goto quit;
    endif

    print ("WAIT...building image list\n");

  // slow build, fast execution, higher memory requirement
    img := file_LoadImageControl("GFX2DEMO.dat", "GFX2DEMO.gci", 1);
  // build image control, returning a pointer to structure allocation

    if (img)
        print("image control=",[HEX] img,"\n");
// show the address of the image control allocation
    else
        putstr("Failed to build image control....\n");
        while(CheckTouchExit() == 0);
// just hang if we didnt get the image list
        r := FAIL;
        goto quit;
    endif

    print ("Loaded ", img[IMG_COUNT], " images\n");
    print ("\nTouch and hold to exit...\n");
```

```
    pause(2000);

    pause(3000);
    gfx_Cls();

    repeat
        n := 0;

        while(n < img[IMG_COUNT] && !exit) // go through all images

            CheckTouchExit();         // if there's a press, exit

            img_SetPosition( img, n, (ABS(RAND() % 240)), (ABS(RAND() % 320)));  // spread out the images

            n++;

        wend

        img_Show(img, ALL);     // update the entire control in 1 hit

    until(exit);

quit:

    mem_Free(img);       // release the image control

    file_Unmount();      // (program must release all resources)

    return r;

endfunc
//=====================================================================
```

2.14.29 file_Mount()

Syntax	**file_Mount();**	
Arguments	**None**	
Returns	**Status**	
	Status	**Returns true if successful.**
Description	Starts up the FAT16 disk file services and allocates a small 32 byte control block for subsequent use. When you open a file using file_Open(..), a further 512 + 44 = 556 bytes are attached to the FAT16 file control block. When you close a file using file_Close(..), the 556 byte allocation is released leaving the 32 byte file control block. The file_Mount() function must be called before any other FAT16 file related functions can be used. The control block and all FAT16 file resources are completely released with file_Unmount().	
Example	`if( !file_Mount() )` `  repeat` `  putstr("Disk not mounted");` `  pause(200);` `  gfx_Cls();` `  pause(200);` `  until( file_Mount() );` `endif`	

2.14.30 file_Unmount()

Syntax	**file_Unmount();**
Arguments	**None**
Returns	**None**
Description	Release any buffers for FAT16 and unmount the Disk File System. This function is to be called to close the FAT16 file system.
Example	`file_Unmount(); //`

2.14.31 file_PlayWAV(fname)

Syntax	**file_PlayWAV(fname);**	
Arguments	**fname**	
	fname	Name of the wav file to be opened and played
Returns	**value**	
	value	**If there are no errors, returns number of blocks to play (1 to 32767)** **If errors occurred, the following is returned** **-7 : Insufficient memory available for WAV buffer and file** **-6 : cant play this rate** **-5 : no data chunk found in first rsector** **-4 : no format data** **-3 : no wave chunk signature** **-2 : bad wave file format** **-1 : file not found**
Description	Open the wav file, decode the header to set the appropriate wave player parameters and set off the playing of the file as a background process. See “Sound Control Functions” for additional play control functions.	
Example	`print("\nding.wav\n");` `  for(n:=0; n<45; n++)` `  pitch := NOTES[n];` `  print([UDEC] pitch,"\r");` `  snd_Pitch(pitch);` `  file_PlayWAV("ding.wav");` `  while(snd_Playing());` `  //pause(500);` `  next`	

2.15. Sound Control Functions

Summary of Functions in this section:

- Snd_Volume(var)
- Snd_Pitch(pitch)
- Snd_BufSize(var)
- Snd_Stop()
- Snd_Pause()
- Snd_Continue()
- Snd_Playing()

2.15.1 snd_Volume(var)

Syntax	**snd_Volume(var);**	
Arguments	**var**	
	var	sound playback volume
	The arguments can be a variable, array element, expression or constant	
Returns	**None**	
Description	Set the sound playback volume. Var must be in the range from 8 (min volume) to 127 (max volume). If var is less than 8, volume is set to 8, and if var > 127 it is set to 127.	
Example	`snd_Volume(127) ; // Set Volume to maximum`	

2.15.2 snd_Pitch(pitch)

Syntax	**snd_Pitch(pitch);**	
Arguments	**pitch**	
	pitch	Sample's playback rate. Minimum is 4KHz. Range is, 4000 – 65535.
	The arguments can be a variable, array element, expression or constant	
Returns	**value**	
	value	**Returns sample's original sample rate.**
Description	Sets the samples playback rate to a different frequency. Setting pitch to zero restores the original sample rate.	
Example	`snd_Pitch(7000); //Play the wav file with a sample frequency of 7KHz.`	

2.15.3 snd_BufSize(var)

Syntax	**snd_BufSize(var);**	
Arguments	**var**	
	var	Specifies the buffer size. 0 = 1024 bytes (default) 1 = 2048 bytes 2 = 4096 bytes
	The arguments can be a variable, array element, expression or constant	
Returns	**None.**	
Description	Specify the a memory chunk size for the wavefile buffer, default size 1024 bytes. Depending on the sample size, memory constraints, and the sample quality, it may be beneficial to change the buffer size from the default size of 1024 bytes. This function is for control of a wav buffer, see the file_PlayWAV(..) ; function	
Example	`snd_BufSize(1);// allocate a 2k wav buffer`	

2.15.4 snd_Stop()

Syntax	**snd_Stop();**
Arguments	**None**
Returns	**None**
Description	Stop any sound that is currently playing, releasing buffers and closing any open wav file. This function is for control of a wav buffer, see the file_PlayWAV(..) ; function
Example	`snd_Stop(); //`

2.15.5 snd_Pause()

Syntax	**snd_Pause();**
Arguments	**None**
Returns	**None**
Description	Pause any sound that is currently playing. This function is for control of a wav buffer, see the file_PlayWAV(..) ; function
Example	`snd_Pause(); //`

2.15.6 snd_Continue()

Syntax	**snd_Continue();**
Arguments	**None**
Returns	**None**
Description	Resume any sound that is currently paused by snd_Pause. This function is for control of a wav buffer, see the file_PlayWAV(..) ; function
Example	`snd_Continue(); //`

2.15.7 snd_Playing()

Syntax	**snd_Playing();**	
Arguments	**None**	
Returns	**value**	
	value	**Number of 512 byte blocks to go.**
Description	Returns 0 if sound has finished playing, else return number of 512 byte blocks to go. This function is for control of a wav buffer, see the file_PlayWAV(..) ; function	
Example	`count := snd_Playing(); //`	

2.16. String Class Functions

Summary of Functions in this section:

- str_Ptr(&var)
- str_GetD(&ptr, &var)
- str_GetW(&ptr, &var)
- str_GetHexW(&ptr, &var)
- str_GetC(&ptr, &var)
- str_GetByte(ptr)
- str_GetWord(ptr)
- str_PutByte(ptr, val)
- str_PutWord(ptr, val)
- str_Match(&ptr, *str)
- str_MatchI(&ptr, *str)
- str_Find(&ptr, *str)
- str_FindI(&ptr, *str)
- str_Length(ptr)
- str_Printf(&ptr, *format)
- str_Cat(&destination, &Source)
- str_CatN(&ptr, str, count)
- str_ByteMove(src, dest, count)
- str_Copy(dest, src)
- str_CopyN(dest, src, count)

2.16.1 str_Ptr(&var)

Syntax	**str_Ptr(&var);**	
Arguments	**var**	
	var	Pointer to string buffer
Returns	**Pointer**	
	Pointer	**Returned value is the byte pointer to string buffer.**
Description	Return a byte pointer to a word region.	
Example	`var buffer[100]; // 200 character buffer for a source string` `var p;  // string pointer` `var n;` `var vars[3];  // for our results` `func main()` `to(buffer); print("0x1234 0b10011001 12345 abacus");` `p := str_Ptr(buffer);//raise string pointer for the string functions` `while(str_GetW(&p, &vars[n++]) != 0); // read all the numbers till we` `//get a non number` `print(vars[0],"\n", vars[1],"\n", vars[2],"\n"); // print them out` `endfunc`	

2.16.2 str_GetD(&ptr, &var)

Syntax	**str_GetD(&ptr, &var);**	
Arguments	**&ptr, &var**	
	ptr	Byte pointer to string.
	var	Destination for our result.
Returns	**Status**	
	Status	
Description	Convert number in a string to DWORD (myvar[2]). NB:- The **address** of the pointer must be passed so the function can advance it if required.	
Example	`var buffer[100]; // 200 character buffer for a source string` `var p;  // string pointer` `var n;` `var vars[6];  // for our results` `func main()` `to(buffer); print("100000 200000 98765432 abacus");` `p := str_Ptr(buffer); // raise a string pointer so we can use the` `// string functions` `while(str_GetD(&p, &vars[n]) != 0) n:=n+2; //read all the numbers` `//till we get a non number` `print( [HEX4] vars[1], ":" , [HEX4] vars[0], "\n" );` `// show the longs as hex numbers` `print( [HEX4] vars[3], ":" , [HEX4] vars[2], "\n" );` `print( [HEX4] vars[5], ":" , [HEX4] vars[4], "\n" );` `endfunc`	

2.16.3 str_GetW(&ptr, &var)

Syntax	**str_GetW(&ptr, &var);**	
Arguments	**&ptr, &var**	
	ptr	byte pointer to string.
	var	destination for our result.
Returns	**Status**	
	Status	**Returns TRUE if function succeeds, advancing ptr.**
Description	Convert number in a string to WORD (myvar). NB:- The address of the pointer must be passed so the function can advance it if required.	
Example	`var buffer[100];  // 200 character buffer for a source string` `var p;  // string pointer` `var n;` `var vars[3];  // for our results` `func main()` `to(buffer); print("0x1234 0b10011001 12345 abacus");` `p := str_Ptr(buffer);  // raise a string pointer so we can use the` `// string functions` `while(str_GetW(&p, &vars[n++]) != 0);  // read all the numbers till` `// we get a non number` `print(vars[0],"\n", vars[1],"\n", vars[2],"\n");  // print them out` `str_Printf (&p, "%s\n" );  // numbers extracted, now just print` `// remainder of string` `endfunc`	

2.16.4 str_GetHexW(&ptr, &var)

Syntax	**str_GetHexW(&ptr, &var);**	
Arguments	**&ptr, &var**	
	ptr	byte pointer to string
	var	destination for our result.
Returns	**Status**	
	Status	Returns TRUE if function succeeds, advancing **ptr**
Description	Convert hex number in a string to WORD (myvar). This function is for extracting 'raw' hex words with no "0x" prefix. Note: The address of the pointer must be passed so the function can advance it if required.	
Example	`var buffer[100];  // 200 character buffer for a source string` `var p;  // string pointer` `var n;` `var vars[4];  // for our results` `func main()` `to(buffer); print("1234 5678 9 ABCD");` `p := str_Ptr(buffer);  // raise a string pointer so we can use the` `//  string functions` `while(str_GetHexW(&p, &vars[n++]) != 0);// read all the hex numbers` `// till we get a non number` `print(vars[0],"\n", vars[1],"\n" , vars[2],"\n", vars[3],"\n");` `endfunc`	

2.16.5 str_GetC(&ptr, &var)

Syntax	**str_GetC(&ptr, &var);**	
Arguments	**&ptr, &var**	
	ptr	Byte pointer to string.
	var	Destination for our result.
	The arguments can be a variable, array element, expression or constant	
Returns	**Status**	
	Status	**Returns TRUE if function succeeds, advancing ptr.**
Description	Get next valid ascii char in a string to myvar. NB:- The address of the pointer must be passed so the function can advance it if required. The function returns 0 if end of string reached. Used for extracting single characters from a string.	
Example	`var p;  // string pointer` `var n;` `var char;` `var buffer[100];  // 200 character buffer for a source string` `func main()` `to(buffer); print("Quick Brown Fox");` `p := str_Ptr(buffer);  // raise a string pointer so we can use the` `//string functions` `while(str_GetC(&p, &char))` `print("p=",p," char is", [CHR] char);  // print characters` `wend` `print("End of string");` `endfunc`	

2.16.6 str_GetByte(ptr)

Syntax	**str_GetByte(ptr);**	
Arguments	**ptr**	
	ptr	Address of byte array or string.
Returns	**byte**	
	byte	**Returns the byte value at pointer location.**
Description	Get a byte to myvar. Similar to "PEEKB" in basic. It is not necessary for byte pointer ptr to be word aligned	
Example	`var buffer[100];    // 200 character buffer for a source string` `var n, p;` `func main()` `to(buffer); print("Testing 1 2 3");` `p := str_Ptr(buffer);        // get a byte pointer from a word region` `n := 0;` `while (n <= str_Length(buffer))` `  print( [HEX2] str_GetByte(p + n++)," ");// print all the chars hex` `                                          // values` `wend` `endfunc`	

2.16.7 str_GetWord(ptr)

Syntax	**str_GetWord(ptr);**	
Arguments	**ptr**	
	ptr	Byte pointer
Returns	**Word**	
	Word	**Returns the word at pointer location.**
Description	Get a word to myvar. Similar to PEEKW in basic. It is not necessary for byte pointer ptr to be word aligned	
Example	`var p;                    // string pointer` `var buffer[10];          // array for 20 bytes` `func main()` `    p := str_Ptr (buffer);   // raise a string pointer` `    str_PutWord (p+3, 100);  // 'poke' the array` `    str_PutWord (p+9, 200);` `    str_PutWord (p+12, 400);` `    print( str_GetWord( p + 3), "\n" );   // 'peek' the array` `    print( str_GetWord( p + 9), "\n" );` `    print( str_GetWord( p + 12), "\n" );` `endfunc`	

2.16.8 str_PutByte(ptr, val)

Syntax	**str_PutByte(ptr, val);**	
Arguments	**ptr, val**	
	ptr	byte pointer to string
	val	byte value to insert.
Returns	**None**	
Description	Put a byte value into a string buffer at ptr Similar to "POKEB" in basic It is not necessary for byte pointer ptr to be word aligned	
Example		

```
var buffer[100];           // 200 character buffer for a source string
var p;                     // string pointer

func main()

p := str_Ptr(buffer);      // raise a string pointer so we can use the
                           // string functions
str_PutByte(p + 3, 'A');   // store some values
str_PutByte(p + 4, 'B');   // store some values
str_PutByte(p + 5, 'C');   // store some values
str_PutByte(p + 7, 'D');   // store some values
str_PutByte(p + 7,  0);    // string terminator
fprint(vars[0],"\n", vars[1],"\n", vars[2],"\n");  // print them out

p := p + 3;                // offset to where we placed the chars
fprint(&p, "%s\n" );       // print the result

// nb, also, understand that the core print service
// assumes a word aligned address so it starts at pos 4
// print( [STR] &buffer[2]);

endfunc
```

2.16.9 str_PutWord(ptr, val)

Syntax	**str_PutWord(ptr, val);**	
Arguments	**Ptr, val**	
	ptr	byte pointer
	val	value to store.
Returns	**None**	
Description	Put a word value into a byte buffer at ptr, similar to "POKEW" in basic. It is not necessary for byte pointer ptr to be word aligned	
Example	`var p;  // string pointer` `var numbers[10];  // array for 20 bytes` `func main()` `  p := str_Ptr (numbers);  // raise a string pointer` `  str_PutWord (p+3, 100);  // 'poke' the array with some numbers` `  str_PutWord (p+9, 200);` `  str_PutWord (p+12, 400);` `  print( str_GetWord( p + 3), "\n" );  // 'peek' the array` `  print( str_GetWord( p + 9), "\n" );` `  print( str_GetWord( p + 12), "\n" );` `endfunc`	

2.16.10 str_Match(&ptr, *str)

Syntax	**str_Match(&ptr, *str);**	
Arguments	**ptr, str**	
	ptr	Address of byte pointer to string buffer.
	str	Pointer string to match.
Returns	**Value**	
	Value	**Returns 0 if no match, else advance ptr to the next position after the match and returns a pointer to the match position.**
Description	Case Sensitive match. Compares the string at position ptr in a string buffer to the string str, skipping over any leading spaces if required. If a match occurs, ptr is advanced to the first position past the match, else ptr is not altered. NB:- The address of the pointer must be passed so the function can advance it if required.	
Example		

```
var buffer[100]; // 200 character buffer for a source string
var p, q; // string pointers
var n;

func main()

    to(buffer); print( " volts 240 " );   // string to parse
    p := str_Ptr(buffer);               // string pointer to be used
                                        // with string functions
    q := p;
    // match the start of the string with "volts"
    if ( n := str_Match( &p, "volts" ) )
        str_Printf ( &p, "%s\n" );  // print remainder of string
    else
        print ( "not found\n" );
    endif
    print ( "startpos=" , q , "\nfindpos=" , n , "\nendpos=" , p );

    repeat
    forever
endfunc
```

2.16.11 str_MatchI(&ptr, *str)

Syntax	**str_MatchI(&ptr, *str);**	
Arguments	**ptr, str**	
	ptr	Address of byte pointer to string buffer.
	str	Pointer string to match.
Returns	**Value**	
	Value	**Returns 0 if no match, else advance ptr to the next position after the match and returns a pointer to the match position**
Description	Case Insensitive match. Compares the string at position ptr in a string buffer to the string str, skipping over any leading spaces if required. If a match occurs, ptr is advanced to the first position past the match, else ptr is not altered. NB:- The address of the pointer must be passed so the function can advance it if required.	
Example		

```
var buffer[100]; // 200 character buffer for a source string
var p, q; // string pointers
var n;

func main()
    // string to parse
    to(buffer); print( "The sun rises in the East" );
    p := str_Ptr(buffer);           // string pointer to be used
                                    // with string functions
    q := p;
    // Will match if the string starts with "The", or "the"
    if ( n := str_MatchI( &p, "the" ) )
        str_Printf ( &p, "%s\n" );    // print remainder of string
    else
        print ( "not found\n" );
    endif
    print ( "startpos=" , q , "\nfindpos=" , n , "\nendpos=" , p );
    repeat
    forever
endfunc
```

2.16.12 str_Find(&ptr, *str)

Syntax	**str_Find(&ptr, *str);**	
Arguments	**ptr, str**	
	ptr	Byte pointer to string buffer.
	str	String to find.
Returns	**Value**	
	Value	**Returns 0 if not found.** **Returns the position of the find if successful.**
Description	Case Sensitive. Searches for string str in string buffer pointed to by ptr. NB:- The pointer ptr is not altered.	
Example		

```
var buffer[100]; // 200 character buffer for a source string
var p; // string pointer
var n;
var strings[4]; // for our test strings

func main()
    txt_Set ( FONT_ID, FONT2 );
    strings[0] := "useful" ;
    strings[1] := "string" ;
    strings[2] := "way" ;
    strings[3] := "class" ;
    to(buffer); print ( "and by the way, the string class is rather
useful " );
    // raise a string pointer so we can use the string functions
    p := str_Ptr(buffer);
    // offset into the buffer a little so we don't see word "way"
    p := p + 13;
    print( "p=" , p , "\n\n" ); // show the start point of our search
    n := 0;
    while ( n < 4 )
        print( "\"" , [STR] strings[n] , "\" is at pos " , str_Find( &p
, strings[n++] ) , "\n" );
    wend
    //note that p is unchanged
    print ( "\nNOTE: p is unchanged, p=" , p );
    repeat
    forever
endfunc
```

2.16.13 str_FindI(&ptr, *str)

Syntax	**str_FindI(&ptr, *str);**	
Arguments	**ptr, str**	
	ptr	Byte pointer to string buffer.
	str	String to find.
Returns	**Value**	
	Value	**Returns 0 if not found.** **Returns the position of the find if successful.**
Description	Case Insensitive. Searches for string str in string buffer pointed to by ptr. NB:- The pointer ptr is not altered.	
Example		

```
var buffer[100]; // 200 character buffer for a source string
var p; // string pointer
var n;
var strings[4]; // for our test strings

func main()
    txt_Set ( FONT_ID, FONT2 );
    strings[0] := "USEFUL" ;
    strings[1] := "string" ;
    strings[2] := "way" ;
    strings[3] := "class" ;
    to(buffer); print ( "and by the way, the String Class is rather
useful " );
    // raise a string pointer so we can use the string functions
    p := str_Ptr(buffer);
    // offset into the buffer a little so we don't see word "way"
    p := p + 13;
    // show the start point of our search
    print( "p=" , p , "\n\n" );
    n := 0;
    while ( n < 4 )
        print( "\"" , [STR] strings[n] , "\" is at pos " , str_FindI (
&p , strings[n++] ) , "\n" );
    wend
    //note that p is unchanged
    print ( "\nNOTE: p is unchanged, p=" , p );
    repeat
    forever
endfunc
```

2.16.14 str_Length(ptr)

Syntax	**str_Length(ptr);**	
Arguments	**ptr**	
	ptr	pointer to string buffer.
Returns	**Value**	
	Value	**Returns String length.**
Description	Returns the length of a string excluding terminator.	
Example		

```
var a;
var b;
var c[40];     // 80 character buffer for a source string
var pa, pc;    //These will be String pointers to a and c[]

func main()

a := mem_Alloc( 200 ); // allocate a dynamic buffer full of random data
mem_Set (a, 'X', 200 );     // fill it full of 'X's
pa := str_Ptr(a);           // raise a string pointer
str_PutByte(pa+20,0);       //Now stick a string terminator in the array
                            //Change the 20 to be between 0 and 199
b := "A string constant" ; // b is a pointer to a string constant

to (c);  print ( "An 'ASCIIZ' string is terminated with a zero" );
pc := str_Ptr(c);           // raise a string pointer so we can use the
                            // string functions
print ("a length:", str_Length(pa), "\n");  // show length of the
                                             // dynamic buffer
print ("b length:", str_Length(b),  "\n");  // show length of the
                                             // static string
print ("c length:", str_Length(pc), "\n");  // show length of the
                                             // 're-directed' string
mem_Free (a);     // test is over, free up the memory
repeat
forever
endfunc
```

2.16.15 str_Printf(&ptr, *format)

Syntax	**str_Printf(&ptr, *format);**	
Arguments	**Ptr, format**	
	ptr	Byte pointer to the input data (structure).
	format	Format string. Note: The address of the pointer must be passed so the function can advance it as required. Note: The format specifier string can be a string pointer, allowing dynamic construction of the printing format. **Format Specifiers:** %c character %s string of characters %d signed decimal %ld long decimal %u unsigned decimal %lu long unsigned decimal %x hex byte %X hex word %lX hex long %b binary word %lb long binary word * indirection prefix (placed after '%' to specify indirect addressing) (number) width description (use between '%' and format specifier to set the field width). Note: If (number) is preceded by 0, the result is Left-pads with zeroes (0) instead of spaces.
Returns	**Pointer**	
	Pointer	**Returns the position of last extraction point. This is useful for processing by other string functions.**
Description	This function prints a formatted string from elements derived from a structured byte region. There is only one input argument, the byte region pointer ptr which is automatically advanced as the format specifier string is processed. The format string is similar to the C language, however, there are a few differences, including the addition of the indirection token * (asterix).	
Example	`var buffer[100];  // 200 character buffer for a source string` `var p, q;  // string pointers` `var n;` `var m[20];  // for our structure example` `var format;  // a pointer to a format string` `func main()` `var k;`	

```
// string print example
to (buffer); print ( "\nHELLO WORLD" );

q := str_Ptr (buffer); // raise a string pointer so we can use the
                       // string functions
p := q;
str_Printf ( &p , "%8s" ); // only prints first 8 characters of
                           // string

putch ('\n');              // new line

p := q;
k := str_Printf ( &p , "%04s" ); // prints 4 leading spaces before
                                 // string

putch ('\n');    // new line
print ( k );     // if required, the return value points to the last
                 // source position and is returned for processing by
                 // other string functions

// print structure elements example,  make a demo structure

n := 0;
m[n++] := "Mrs Smith" ;
m[n++] := 200 ;
m[n++] := 300 ;
m[n++] := 0xAA55 ;
m[n++] := 500 ;

// make a demo format control string

format := "%*s\n%d\n%d\n%016b\n%04X" ; // format string for printing
                                       // structure m

// print the structure in the required format

p := str_Ptr (m);           // point to structure m
str_Printf (&p, format);    // use the format string to print the
                            // structure

endfunc
```

2.16.16 str_Cat(&destination, &source)

Syntax	**str_Cat(&destination, &source);**	
Arguments	**destination, source**	
	destination	Destination string address
	source	Source string address
Returns	**Pointer**	
	Pointer	**Returns pointer to the destination.**
Description	Appends a copy of the source string to the destination string. The terminating null character in destination is overwritten by the first character of source, and a new null-character is appended at the end of the new string formed by the concatenation of both in destination.	
Example	`var buf[100]; // 200 character buffer for a source string` `func main()` `  var p ;` `  to(buf) ;` `  print("Hello ") ;` `  p := str_Ptr(buf) ;` `  str_Cat(p,"There"); // Will append "There" to the end of buf` `  print([STR] buf) ;` `  repeat` `  forever` `endfunc`	

2.16.17 str_CatN(&ptr, str, count)

Syntax	**str_CatN(&ptr, str, count);**	
Arguments	**ptr, str, count**	
	ptr	Destination string address
	str	Source string address
	count	Number of characters to be concatenated.
Returns	**Pointer**	
	Pointer	**Returns pointer to the destination.**
Description	The number of characters copied is limited by "count". The terminating null character in destination is overwritten by the first character of source, and a new null-character is appended at the end of the new string formed by the concatenation of both in destination.	
Example	`var buf[100]; // 200 character buffer for a source string` `func main()` `  var p ;` `  to(buf) ;` `  print("Sun ") ;` `  p := str_Ptr(buf) ;` `  str_CatN(p,"Monday",3); // Concatenate "Mon" to the end of buf` `  print([STR] buf) ;` `  repeat` `  forever` `endfunc`	

2.16.18 str_ByteMove(src, dest, count)

Syntax	**str_ByteMove(src, dest, count);**	
Arguments	**src, dest, count**	
	src	points to byte aligned source.
	dest	points to byte aligned destination.
	count	Number of bytes to transfer.
Returns	**Pointer**	
	Pointer	Returns a pointer to the end of the destination (which is "dest" + "count").
Description	Copy bytes from "src" to "dest", stopping only when "count" is exhausted. No terminator is appended, it is purely a byte copy, and any zeroes encountered will also be copied.	
Example	var src, dest, mybuf1[10], mybuf2[10]; // string pointers and two 20 byte buffers to(mybuf1); putstr("TESTING 123"); src := strPtr(mybuf1); dest := str_Ptr(mybuf2); src += 6; // move src pointer to "G 123" str_ByteMove(src, dest, 6); // move to second buffer (including the zero terminator) putstr(mybuf2); // print result nextpos := str_ByteMove(s, d, 100);	

2.16.19 str_Copy(dest, src)

Syntax	**str_Copy(dest, src);**	
Arguments	**dest, src**	
	dest	points to byte aligned destination.
	src	points to byte aligned source.
Returns	**Pointer**	
	Pointer	Returns a pointer to the 0x00 string terminator at the end of "dest" (which is "dest" + str_Length(src);).
Description	Copy a string from "src" to "dest", stopping only when the end of source string "src" is encountered (0x00 terminator). The terminator is always appended, even if "src" is an empty string.	
Example	`nextplace := str_Copy(d, s);`	

2.16.20 str_CopyN(dest, src, count)

Syntax	**str_CopyN(dest, src, count);**	
Arguments	**dest, src, count**	
	dest	points to byte aligned destination.
	src	points to byte aligned source.
	count	Maximum number of bytes to copy.
Returns	**Pointer**	
	Pointer	Returns a pointer to the 0x00 string terminator at the end of "dest" (which is "dest" + str_Length(src);).
Description	Copy a string from "src" to "dest", stopping only when "count" is exhausted, or end of source string "str" is encountered (0x00 string terminator). The terminator is always appended, even if "count" is zero, or "src" is a null string.	
Example	`nextplace := str_CopyN(d, s, 100);`	

2.17. Touch Screen Functions

Summary of Functions in this section:

- touch_DetectRegion(x1, y1, x2, y2)
- touch_Set(mode)
- touch_Get(mode)

Note: Touch Screen functions do not apply to uVGA-II/III modules.

2.17.1 touch_DetectRegion(x1, y1, x2, y2)

Syntax	**touch_DetectRegion(x1, y1, x2, y2);**	
Arguments	**X1, y1, x2, y2**	
	x1	specifies the horizontal position of the top left corner of the region.
	y1	specifies the vertical position of the top left corner of the region.
	x2	specifies the horizontal position of the bottom right corner of the region.
	y2	specifies the vertical position of the bottom right corner of the region.
Returns	**None**	
Description	Specifies a new touch detect region on the screen. This setting will filter out any touch activity outside the region and only touch activity within that region will be reported by the status poll touch_Get(0); function.	
Example	`gfx_Rectangle(100, 100, 201, 201, YELLOW); // draw a rectangle with //a yellow border` `touch_DetectRegion(101, 101, 200, 200); // limit touch detect region to //within the rectangle`	

2.17.2 touch_Set(mode)

Syntax	**touch_Set(mode);**	
Arguments	**mode**	
	mode	**mode = 0 :** Enable Touch Screen touch_Set(0); Enables and initialises Touch Screen hardware **mode = 1 :** Disable Touch Screen touch_Set(1); Disables the Touch Screen. Note: Touch Screen task runs in the background and disabling it when not in use will free up extra resources for 4DGL CPU cycles. **mode = 2 :** Default Touch Region touch_Set(2); This will reset the current active region to default which is the full screen area
Returns	**None**	
Description	Sets various Sets various Touch Screen related parameters.	
Example	`touch_Set(TOUCH_ENABLE); // .`	

2.17.3 touch_Get(mode)

Syntax	**touch_Get(mode);**	
Arguments	**mode**	
	mode	**mode = 0 :** Get Status **mode = 1 :** Get X coordinates **mode = 2 :** Get Y coordinates
Returns	**Value**	
	Value	**mode = 0** Returns the various states of the touch screen 0 = INVALID/NOTOUCH 1 = PRESS 2 = RELEASE 3 = MOVING **mode = 1 :** Returns the X coordinates of the touch reported by mode 0 **mode = 2 :** Returns the Y coordinates of the touch reported by mode 0
Description	Returns various Touch Screen parameters to caller.	
Example		

```
state := touch_Get(TOUCH_STATUS);  // get touchscreen status
        x := touch_Get(TOUCH_GETX);
        y := touch_Get(TOUCH_GETY);

        if (state == TOUCH_PRESSED)   // see if Exit hit
            if ( x > 170 && y > 280 ) // EXIT button
                gfx_Cls();
                exit := -1;
            endif

            if (vertical)
                if ( x > 170 && (y > 240 && y < 270 ))// Horiz button
                    vertical := 0;
                    exit := 1;
                endif
            else
                if ( x > 170 && (y > 200 && y < 230 ))// Vert button
                    vertical := 1;
                    exit := 2;
                endif
            endif

        endif
```

2.18. Image Control Functions

Summary of Functions in this section:

- img_SetPosition(handle, index, xpos, ypos)
- img_Enable(handle, index)
- img_Disable(handle, index)
- img_Darken(handle, index)
- img_Lighten(handle, index)
- img_SetWord(handle, index, offset, word)
- img_GetWord(handle, index, offset)
- img_Show(handle, index)
- img_SetAttributes(handle, index, value)
- img_ClearAttributes(handle, index, value)
- img_Touched(handle, index)

2.18.1 img_SetPosition(handle, index, xpos, ypos)

Syntax	**img_SetPosition(handle, index, xpos, ypos);**	
Arguments	**handle, index, xpos, ypos**	
	handle	Pointer to the Image List.
	index	Index of the images in the list.
	xpos	Top left horizontal screen position where image is to be displayed.
	ypos	Top left vertical screen position where image is to be displayed.
Returns	**Status**	
	Status	**True or False**
Description	This function requires that an image control has been created with the file_LoadImageControl(...); function. Sets the position where the image will next be displayed. Returns TRUE if index was ok and function was successful. (the return value is usually ignored). You may turn off an image so when img_Show() is called, the image will not be shown. This function requires that an image control has been created with the file_LoadImageControl(...); function.	
Example	`// make a simple 'window'` `gfx_Panel(PANEL_RAISED, 0, 0, 239, 239, GRAY);` `img_SetPosition(Ihndl, BTN_EXIT,  224,2);//set checkout box position` `img_Enable(Ihndl, BTN_EXIT);  //enable checkout box`	

2.18.2 img_Enable(handle, index)

Syntax	**img_Enable(handle, index);**	
Arguments	**handle, index**	
	handle	Pointer to the Image List.
	index	Index of the images in the list.
Returns	**Status**	
	Status	**TRUE or FALSE.**
Description	This function requires that an image control has been created with the file_LoadImageControl(...); function. Enables a selected image in the image list. Returns TRUE if index was ok and function was successful. This is the default state so when img_Show() is called all the images in the list will be shown. To enable all of the images in the list at the same time set index to -1. To enable a selected image, use the image index number.	
Example	`r := img_Enable(hImageList, imagenum);//`	

2.18.3 img_Disable(handle, index)

Syntax	**img_Disable(handle, index);**	
Arguments	**handle, index**	
	handle	Pointer to the Image List.
	index	Index of the images in the list.
Returns	**Status**	
	Status	**TRUE or FALSE**
Description	This function requires that an image control has been created with the file_LoadImageControl(...); function. Disables an image in the image list. Returns TRUE if index was ok and function was successful. Use this function to turn off an image so that when img_Show() is called the selected image in the list will not be shown. To disable all of the images in the list at the same time set index to -1.	
Example	`r := img_Disable(hImageList, imagenum);//`	

2.18.4 img_Darken(handle, index)

Syntax	**img_Darken(handle, index);**	
Arguments	**handle, index**	
	handle	Pointer to the Image List.
	index	Index of the images in the list.
Returns	**Status**	
	Status	**TRUE or FALSE**
Description	This function requires that an image control has been created with the file_LoadImageControl(...); function. Darken an image in the image list. Returns TRUE if index was ok and function was successful. Use this function to darken an image so that when img_Show() is called the control will take effect. To darken all of the images in the list at the same time set index to -1. Note: This feature will take effect one time only and when img_Show() is called again the darkened image will revert back to normal.	
Example	`r := img_Darken(hImageList, imagenum);`	

2.18.5 img_Lighten(handle, index)

Syntax	**img_Lighten(handle, index);**	
Arguments	**handle, index**	
	handle	Pointer to the Image List.
	index	Index of the images in the list.
Returns	**Status**	
	Status	**TRUE or FALSE**
Description	This function requires that an image control has been created with the file_LoadImageControl(...); function. Lighten an image in the image list. Returns TRUE if index was ok and function was successful. Use this function to lighten an image so that when img_Show() is called the control will take effect. To lighten all of the images in the list at the same time set index to -1. Note: This feature will take effect one time only and when img_Show() is called again the lightened image will revert back to normal.	
Example	`r := img_Lighten(hImageList, imagenum);`	

2.18.6 img_SetWord(handle, index, offset, word)

Syntax	**img_SetWord(handle, index, offset, word);**	
Arguments	**handle, index**	
	handle	Pointer to the Image List.
	index	Index of the images in the list.
	offset	Offset of the required word in the image entry
	word	The word to be written to the entry
Returns	**Status**	
	Status	**TRUE or FALSE**
Description	This function requires that an image control has been created with the file_LoadImageControl(...); function. Set specified word in an image entry. Returns TRUE if successful, return value usually ignored. IMAGE_XPOS 2 // WORD image location X IMAGE_YPOS 3 // WORD image location Y IMAGE_FLAGS 6 // WORD image flags IMAGE_DELAY 7 // WORD inter frame delay IMAGE_INDEX 9 // WORD current frame IMAGE_TAG 12 // WORD user variable #1 IMAGE_TAG2 13 // WORD user variable #2 **Note:** Not all Constants are listed as some are Read Only. img_Show(..) will now show error box for out of range video frames. Also, if frame is set to -1, just a rectangle will be drawn in background colour to blank an image. It applies to PmmC R29 or above.	
Example	`func cat()` `var private frame := 0; // start with frame 0` `var private image := SPRITE_CAT; // cat image, can be changed with` `// cat.image := xxx` `var private speed := 30;` `img_SetWord(Ihndl, image, IMAGE_INDEX, frame++);` `frame := frame % img_GetWord(Ihndl, image, IMAGE_FRAMES);` `img_Show(Ihndl, image);` `sys_SetTimer(TIMER3,speed); // reset the event timer` `endfunc`	

2.18.7 img_GetWord(handle, index, offset)

Syntax	**img_GetWord(handle, index, offset);**	
Arguments	**handle, index**	
	handle	Pointer to the Image List.
	index	Index of the images in the list.
	offset	Offset of the required word in the image entry
Returns	**Value**	
	value	**Returns the image entry in the list.**
Description	This function requires that an image control has been created with the file_LoadImageControl(...); function. Returns specified word from an image entry. IMAGE_LOWORD 0 // WORD image address LO IMAGE_HIWORD 1 // WORD image address HI IMAGE_XPOS 2 // WORD image location X IMAGE_YPOS 3 // WORD image location Y IMAGE_WIDTH 4 // WORD image width IMAGE_HEIGHT 5 // WORD image height IMAGE_FLAGS 6 // WORD image flags IMAGE_DELAY 7 // WORD inter frame delay IMAGE_FRAMES 8 // WORD number of frames IMAGE_INDEX 9 // WORD current frame IMAGE_CLUSTER 10 // WORD image start cluster pos (for FAT16 only) IMAGE_SECTOR 11 // WORD image start sector in cluster pos (for FAT16 only) IMAGE_TAG 12 // WORD user variable #1 IMAGE_TAG2 13 // WORD user variable #2	
Example	`myvar := img_GetWord(hndl, 5, IMAGE_YPOS); //`	

2.18.8 img_Show(handle, index)

Syntax	**img_Show(handle, index);**	
Arguments	**handle, index**	
	handle	Pointer to the Image List.
	index	Index of the images in the list.
Returns	**Status**	
	Status	**TRUE or FALSE.**
Description	This function requires that an image control has been created with the file_LoadImageControl(...); function. Enable the displaying of the image entry in the image control. Returns TRUE if successful but return value is usually ignored.	
Example	`img_Show(hImageList, imagenum);`	

2.18.9 img_SetAttributes(handle, index, value)

Syntax	**img_SetAttributes(handle, index, value);**	
Arguments	**handle, index, value**	
	handle	Pointer to the Image List.
	index	Index of the images in the list.
	value	Refers to various bits in the image control entry (see image attribute flags)
Returns	**Status**	
	Status	**TRUE or FALSE**
Description	This function SETS one or more bits in the IMAGE_FLAGS field of an image control entry. "value" refers to various bits in the image control entry (see image attribute flags). A '1' bit in the "value" field SETS the respective bit in the IMAGE_FLAGS field of the image control entry. I_ENABLED 0x8000 // bit 15, set for image enabled I_DARKEN 0x4000 // bit 14, display dimmed I_LIGHTEN 0x2000 // bit 13, display bright I_TOUCHED 0x1000 // bit 12, touch test result I_Y_LOCK 0x0800 // bit 11, stop Y movement I_X_LOCK 0x0400 // bit 10, stop X movement I_TOPMOST 0x0200 // bit 9, draw on top of other images next update I_STAYONTOP 0x0100 // bit 8, draw on top of other images always I_TOUCH_DISABLE 0x0020 // bit 5, set to disable touch for this image, default=1 for movie, 0 for image img_ClearAttributes(handle, index, value);	
Example	: : `img_Enable(Ihndl, SPRITE_CAT); // we'll also use small cat video` `img_SetAttributes(Ihndl, SPRITE_CAT, I_NOGROUP);` `img_SetPosition(Ihndl, SPRITE_CAT, 160, 180); // set its position` :	

2.18.10 img_ClearAttributes(handle, index, value)

Syntax	**img_ClearAttributes(handle, index, value);**	
Arguments	**handle, index, value**	
	handle	Pointer to the Image List.
	index	Index of the images in the list.
	value	a '1' bit indicates that a bit should be set and a '0' bit indicates that a bit is not altered. Note: if index is set to -1, the attribute is altered in ALL of the entries in the image list . The constant ALL is set to -1 specifically for this purpose.
Returns	**Status**	
	Status	**TRUE or FALSE**
Description	Clear various image attribute flags in a image control entry. (see image attribute flags below) Image attribute flags may be combined with the + or | operators, eg:- img_ClearAttributes(hndl, ALL, I_Y_LOCK | I_X_LOCK); // allow all images to move in any direction This function requires that an image control has been created with the file_LoadImageControl(...); function. Returns TRUE if index was ok and function was successful. (the return value is usually ignored). Image attribute flags I_ENABLED 0x8000 // bit 15, set for image enabled I_DARKEN 0x4000 // bit 14, display dimmed I_LIGHTEN 0x2000 // bit 13, display bright I_TOUCHED 0x1000 // bit 12, touch test result I_Y_LOCK 0x0800 // bit 11, stop Y movement I_X_LOCK 0x0400 // bit 10, stop X movement I_TOPMOST 0x0200 // bit 9, draw on top of other images next update I_STAYONTOP 0x0100 // bit 8, draw on top of other images always I_TOUCH_DISABLE 0x0020 // bit 5, set to disable touch for this image, default=1 for movie, 0 for image	
Example	`img_ClearAttributes(hndl, 5, value ); //`	

2.18.11 img_Touched(handle, index)

Syntax	**img_Touched(handle, index);**	
Arguments	**handle, index**	
	handle	Pointer to the Image List.
	index	Index of the images in the list.
Returns	**Status**	
	Status	**Returns index or -1.**
Description	This function requires that an image control has been created with the file_LoadImageControl(...); function. Returns index if image touched or returns -1 if no image was touched. If index is passed as -1 the function tests all images and returns -1 if no image was touched or returns index.	
Example	`if(state ==  TOUCH_PRESSED)` ` n := img_Touched(Ihndl, -1);//scan image list, looking for a touch` `  if(n != -1)` `  last := n;` `  button := n;` `  img_Lighten(Ihndl, n);//lighten the button touched` `  img_Show(Ihndl, -1);  // restore the images` `  endif` `endif`	

2.19. Memory Allocation Functions

Summary of Functions in this section:

- mem_Alloc(size)
- mem_Allocv(size)
- mem_Allocz(size)
- mem_Realloc(ptr, size)
- mem_Free(allocation)
- mem_Heap()
- mem_Set(ptr, char, size)
- mem_Copy(source, destination, count)
- mem_Compare(ptr1, ptr2, count)

2.19.1 mem_Alloc(size)

Syntax	**mem_Alloc(size);**	
Arguments	**size (byte)**	
	size	Specifies the number of bytes that's allocated from the heap.
Returns	**value**	
	value	**Returned value is the pointer (Word) to the allocation if successful.** **If function fails returns a null (0).**
Description	Allocate a block of memory to pointer myvar. The allocated memory contains garbage but is a fast allocation. The block must later be released with mem_Free(myvar);	
Example	`myvar := mem_Alloc(100); //`	

2.19.2 mem_AllocV(size)

Syntax	**mem_AllocV(size);**	
Arguments	**size (Byte)**	
	size	Specifies the number of bytes that's allocated from the heap.
Returns	**Value**	
	Value	**Returned value is the pointer (Word) to the allocation if successful. If function fails returns a null (0).**
Description	Allocate a block of memory to pointer myvar. The block of memory is filled with initial signature values. The block starts with A5,5A then fills with incrementing number eg:- A5,5A,00,01,02,03...FF,00,11.... This can be helpful when debugging. The block must later be released with mem_Free(myvar).	
Example	`myvar := mem_AllocV(100);//`	

2.19.3 mem_Allocz(size)

Syntax	**mem_Allocz(size);**	
Arguments	**size**	
	size	Specifies the number of bytes that's allocated from the heap.
Returns	**Value**	
	Value	**Returned value is the pointer to the allocation if successful. If function fails returns a null (0).**
Description	Allocate a block of memory to pointer myvar. The block of memory is filled with zeros. The block must later be released with mem_Free(myvar);	
Example	`myvar := mem_Allocz(100);//`	

2.19.4 mem_Realloc(&ptr, size)

Syntax	**mem_Realloc(&ptr, size);**	
Arguments	**ptr, size**	
	ptr	specifies the new location to reallocate the memory block.
	size	specifies the number of bytes of the block.
Returns	**Status**	
	Status	**See the Description.**
Description	The function may move the memory block to a new location, in which case the new location is returned. The content of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved. If the new size is larger, the value of the newly allocated portion is indeterminate. In case that ptr is NULL, the function behaves exactly as mem_Alloc(), assigning a new block of size bytes and returning a pointer to the beginning of it. In case that the size is 0, the memory previously allocated in ptr is deallocated as if a call to mem_Free(myvar)was made, and a NULL pointer is returned.	
Example	`myvar := mem_Realloc(myptr, 100); //`	

2.19.5 mem_Free(allocation)

Syntax	**mem_Free(allocation);**	
Arguments	**allocation**	
	allocation	specifies the location of memory block to free up.
Returns	**Status**	
	Status	Returns non-zero if function is successful Returns 0 if the function fails.
Description	The function de-allocates a block of memory previously created with mem_Alloc(...), mem_AllocV(...) or mem_AllocZ(...).	
Example	`test := mem_Free(myvar); //`	

2.19.6 mem_Heap()

Syntax	**mem_Heap();**	
Arguments	**None**	
Returns	**Value**	
	Value	**Returns the largest available memory chunk of the heap.**
Description	Returns byte size of the largest chunk of memory available in the heap.	
Example	`howmuch := mem_Heap();`	

2.19.7 mem_Set(ptr, char, size)

Syntax	**mem_Set(ptr, char, size);**	
Arguments	**ptr, char, size**	
	ptr	specifies the memory block.
	char	specifies the value to fill the block with.
	size	specifies the size of the block in Bytes.
Returns	**Pointer**	
	Pointer	**Returns the pointer.**
Description	Fill a block of memory with a byte value.	
Example		

```
var mybuf[5];
var i;

func main()

mem_Set(mybuf,0x55,5);  //Only fills half of mybuf[]
for(i:=0;i<sizeof(mybuf);i++)   //Show what is in the buffer
      print(" 0x",[HEX]mybuf[i]);
next
mem_Set(mybuf,0xAA,sizeof(mybuf)*2); //Fill entire buffer
print("\n"); //New line
for(i:=0;i<sizeof(mybuf);i++)
      print(" 0x",[HEX]mybuf[i]);
next
repeat
forever
```

2.19.8 mem_Copy(source, destination, count)

Syntax	**mem_Copy(source, destination, count);**	
Arguments	**source, destination, count**	
	source	specifies the source memory block.
	destination	specifies the destination memory block.
	count	specifies the size of the blocks.
Returns	**Pointer**	
	Pointer	**Returns source.**
Description	Copy a block of memory from source to destination. Note: src can be a string constant eg:- myptr := mem_Copy("TEST STRING", ptr2, 12);	
Example	`myptr := mem_Copy(ptr1, ptr2, 100); //`	

2.19.9 mem_Compare(ptr1, ptr2, count)

Syntax	**mem_Compare(ptr1, ptr2, count);**	
Arguments	**ptr1, ptr2, count**	
	ptr1	specifies the 1st memory block.
	ptr2	specifies the 2nd memory block.
	count	specifies the number of bytes to compare.
Returns	**Value**	
	Value	**Returns 0 if we have a match, -1 if ptr1 < ptr2, and +1 if ptr2 > ptr1. (The comparison is done alphabetically)**
Description	Compare two blocks of memory **ptr1** and **ptr2.**	
Example	`test := mem_Compare(this_block, that_block, 100); //`	

2.20. General Purpose Functions

Summary of Functions in this section:

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

2.20.1 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 (status)  // if fire button pressed` `  pause(30)  // slow down the loop` `else` `  ...`	

2.20.2 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
```

2.20.3 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. Picaso EVE System Registers Memory Map

The following tables outline in detail the Picaso system registers and flags.

Table 3.1: WORD-Size Registers Memory Map

LABEL	ADDRESS		USAGE
	DEC	HEX	
RANDOM_LO	32	0x20	random generator LO word
RANDOM_HI	33	0x21	random generator HI word
SYSTEM_TIMER_LO	34	0x22	1msec system timer LO word
SYSTEM_TIMER_HI	35	0x23	1msec system timer HI word
TIMER0	36	0x24	1msec user timer 0
TIMER1	37	0x25	1msec user timer 1
TIMER2	38	0x26	1msec user timer 2
TIMER3	39	0x27	1msec user timer 3
TIMER4	40	0x28	1msec user timer 3
TIMER5	41	0x29	1msec user timer 3
TIMER6	42	0x2A	1msec user timer 3
TIMER7	43	0x2B	1msec user timer 3
SYS_X_MAX	44	0x2C	display hardware X res-1
SYS_Y_MAX	45	0x2D	display hardware Y res-1
GFX_XMAX	46	0x2E	width of current orientation
GFX_YMAX	47	0x2F	height of current orientation
GFX_LEFT	48	0x30	image left real point
GFX_TOP	49	0x31	image top real point
GFX_RIGHT	50	0x32	image right real point
GFX_BOTTOM	51	0x33	image bottom real point
GFX_X1	52	0x34	image left clipped point
GFX_Y1	53	0x35	image top clipped point
GFX_X2	54	0x36	image right clipped point
GFX_Y2	55	0x37	image bottom clipped point
GFX_X_ORG	56	0x38	current X origin
GFX_Y_ORG	57	0x39	current Y origin
GFX_HILITE_LINE	58	0x3A	current multi line button hilite line
GFX_LINE_COUNT	59	0x3B	count of lines in multiline button
GFX_LAST_SELECTION	60	0x3C	Last selected line
GFX_HILIGHT_BACKGROUND	61	0x3D	multi button hilite background colour
GFX_HILIGHT_FOREGROUND	62	0x3E	multi button hilite background colour
GFX_BUTTON_FOREGROUND	63	0x3F	store default text colour for hilite line tracker
GFX_BUTTON_BACKGROUND	64	0x40	store default button colour for hilite line tracker
GFX_BUTTON_MODE	65	0x41	store current buttons mode
GFX_TOOLBAR_HEIGHT	66	0x42	height above
GFX_STATUSBAR_HEIGHT	67	0x43	height below
GFX_LEFT_GUTTER_WIDTH	68	0x44	width to left
GFX_RIGHT_GUTTER_WIDTH	69	0x45	width to right
GFX_PIXEL_SHIFT	70	0x46	pixel shift for button depress illusion
GFX_VECT_X1	71	0x47	gp rect, used by multiline button to hilite required line
GFX_VECT_Y1	72	0x48	
GFX_VECT_X2	73	0x49	
GFX_VECT_Y2	74	0x4A	
GFX_THUMB_PERCENT	75	0x4B	size of slider thumb as percentage
GFX_THUMB_BORDER_DARK	76	0x4C	darker shadow of thumb
GFX_THUMB_BORDER_LIGHT	77	0x4D	lighter shadow of thumb
TOUCH_XMINCAL	78	0x4E	touch calibration value

TOUCH_YMINCAL	79	0x4F	touch calibration value
TOUCH_XMAXCAL	80	0x50	touch calibration value
TOUCH_YMAXCAL	81	0x51	touch calibration value
IMG_WIDTH	82	0x52	width of currently loaded image
IMG_HEIGHT	83	0x53	height of currently loaded image
IMG_FRAME_DELAY	84	0x54	if image, else inter frame delay for movie
IMG_FLAGS	85	0x55	bit 4 determines colour mode, other bits reserved
IMG_FRAME_COUNT	86	0x56	count of frames in a movie
IMG_PIXEL_COUNT_LO	87	0x57	count of pixels in the current frame
IMG_PIXEL_COUNT_HI	88	0x58	count of pixels in the current frame
IMG_CURRENT_FRAME	89	0x59	last frame shown
MEDIA_ADDRESS_LO	90	0x5A	uSD byte address LO
MEDIA_ADDRESS_HI	91	0x5B	uSD byte address HI
MEDIA_SECTOR_LO	92	0x5C	uSD sector address LO
MEDIA_SECTOR_HI	93	0x5D	uSD sector address HI
MEDIA_SECTOR_COUNT	94	0x5E	uSD number of bytes remaining in sector
TEXT_XPOS	95	0x5F	text current x pixel position
TEXT_YPOS	96	0x60	text current y pixel position
TEXT_MARGIN	97	0x61	text left pixel pos for carriage return
TXT_FONT_TYPE	98	0x62	font type, 0 = system font, else pointer to user font
TXT_FONT_MAX	99	0x63	max number of chars in font
TXT_FONT_OFFSET	100	0x64	starting offset (normally 0x20)
TXT_FONT_WIDTH	101	0x65	current font width
TXT_FONT_HEIGHT	102	0x66	Current font height
GFX_TOUCH_REGION_X1	103	0x67	touch capture region
GFX_TOUCH_REGION_Y	104	0x68	touch capture region
GFX_TOUCH_REGION_X2	105	0x69	touch capture region
GFX_TOUCH_REGION_Y2	106	0x6A	touch capture region
GFX_CLIP_LEFT_VAL	107	0x6B	left clipping point (set with gfx_ClipWindow(...)
GFX_CLIP_TOP_VAL	108	0x6C	top clipping point (set with gfx_ClipWindow(...)
GFX_CLIP_RIGHT_VAL	109	0x6D	right clipping point (set with gfx_ClipWindow(...)
GFX_CLIP_BOTTOM_VAL	110	0x6E	bottom clipping point (set with gfx_ClipWindow(...)
GFX_CLIP_LEFT	111	0x6F	current clip value (reads full size if clipping turned off)
GFX_CLIP_TOP	112	0x70	current clip value (reads full size if clipping turned off)
GFX_CLIP_RIGHT	113	0x71	current clip value (reads full size if clipping turned off)
GFX_CLIP_BOTTOM	114	0x72	current clip value (reads full size if clipping turned off)
GRAM_PIXEL_COUNT_LO	115	0x73	LO word of count of pixels in the set GRAM area
GRAM_PIXEL_COUNT_HI	116	0x74	HI word of count of pixels in the set GRAM area
TOUCH_RAW_X	117	0x75	12 bit raw A2D X value from touch screen
TOUCH_RAW_Y	118	0x76	12 bit raw A2D Y value from touch screen
GFX_LAST_CHAR_WIDTH	119	0x77	calculated char width from last call to charWidth function
GFX_LAST_CHAR_HEIGHT	120	0x78	calculated height from last call to charHeight function
GFX_LAST_STR_WIDTH	121	0x79	calculated width from last call to strWidth function
GFX_LAST_STR_HEIGHT	122	0x7A	calculated height from last call to strHeight function
NOTE: These registers are accessible with **peekW** and **pokeW** functions.			

4. Appendix A : Example 4DGL Code

```
#platform "uOLED-32028-P1_GFX2"

/************************************************
* Filename: Window.4dg
* Created: 2010/06/17
* Author: 4D
* Description: A simple window object example
*
*
* NB:- This program should be written to flash so
* it becomes the top down program.
*
************************************************/

#inherit "4DGL_16bitColours.fnc"
//#inherit "FONT4.fnt"

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

#STACK 500          // make sure stack is big enough for main prog and called functions

// colour scheme
#CONST
    WINDOW_COLOR        GRAY
    TITLEBAR_COLOR      NAVY
    TITLETXT_COLOR      CYAN
    STATUSBAR_COLOR     GRAY
    STATUSTXT_COLOR     YELLOW
#END

//---------------------------------------------------------------------------
// local global variables
//---------------------------------------------------------------------------
var D;              // pointer to disk struct
                    // (we keep 2 copies so we can test for a state -
var tempstr[20];  // general purpose string, 40 bytes

//===========================================================================
// button texts
//===========================================================================
#DATA
    word buttontexts tst1, tst2, tst3, tst4, btnexit
    byte tst1 "TEST1\0"
    byte tst2 "TEST2\0"
    byte tst3 "TEST3\0"
    byte tst4 "TEST4\0"
    byte btnexit "_\0"
#END

//===========================================================================
// In the main function, we establish a simple window and activate it.
// From then on, it is a simple matter of polling the window, and acting
// on the number (message) it returns. This greatly simplifies the
// application, as all touch testing is handled by the window itself.
//===========================================================================
func main()
    var Wmsg;             // message from window

    gfx_Cls();
    txt_Set(FONT_ID, FONT3);
    print("Memory available = ",mem_Heap(),"\n"); //show biggest chunk we have

    // set some window properties
    aWindow.title := "A Test Window";
```

```
    aWindow.xpos := 10;
    aWindow.ypos := 60;
    aWindow.font := FONT2;
    aWindow(INITIALIZE);          // draw window / buttons for the first time

    pause(500);
    aWindow.statusbartext := init_Drive(); //mount the disk, setting status msg

    aWindow(REDRAW_STATUS);    // update the status bar

    // now just stay in a loop, getting info from window
    repeat
        Wmsg := aWindow(SCAN); // scan for any changes

        // if return value non zero, its a button number
        if(Wmsg)

            switch
                //-------------------------------------------------------
                // would normall do some exit action here
                // but for demo we just reset the window
                //-------------------------------------------------------
                case (Wmsg == MAXBUTTONS)  // if it was the exit (last) button

                    aWindow(INITIALIZE);
                    break;

                //-------------------------------------------------------
                // update status and title with the button number
                //-------------------------------------------------------
                case (Wmsg >0 && Wmsg < MAXBUTTONS)
                    to(tempstr); print("Button #",Wmsg);
                      // print return value to the temp buffer
                    to(APPEND); putstr("         ");
                   // clear string tail
                    aWindow.statusbartext := tempstr;
                   // use temp buffer for status text
                    aWindow(REDRAW_STATUS);
                   // update the status bar

                    to(tempstr); putstr(buttontexts[Wmsg-1]);
                   // print the button text to the temp buffer
                    to(APPEND); putstr("         ");
                    // clear string tail
                    aWindow.title := tempstr;
                   // use temp buffer for title text
                    aWindow(REDRAW_TITLE);
                   // update the title bar
                    break;

                default:

            endswitch

        endif
    forever
endfunc

#constant MAXBUTTONS 5      // 4 pushbuttons

// enumeration for the window's sub functions
#constant INITIALIZE, UPDATE, SCAN, REDRAW_TITLE, REDRAW_STATUS

// example for a simple Window object
func aWindow(var subfunc)

    // window properties
    var private xpos, ypos;                          // window position
```

```
    var private windowWidth := 200;                  // default window width
    var private windowHeight := 200;                 // default window height
    var private windowcolor := WINDOW_COLOR;         // default window colour

    // title bar properties
    // title bar text pointer, set default title bar text
    var private title;                  // title bar text pointer
    var private font := FONT3;                       // default font
    var private titlebarheight := 20;                // default title bar height
    var private titlebarcolor := TITLEBAR_COLOR;     // default title bar colour
    var private titletextcolor := TITLETXT_COLOR;    // default title text colour

    // status bar properties
    var private statusbartext;                       // status bar text pointer
    var private statusbarheight := 10;               // default status bar height
    var private statusbarcolor := STATUSBAR_COLOR;   // default status bar colour
    var private statustextcolor := STATUSTXT_COLOR; // default status txt colour

    var private btncolor := LIGHTGREY;               // default button colour
    var private textcolor := BLACK;                  // default text colour
    var private ygap := 4;                        // vertical gap between buttons
    var private touchX1[MAXBUTTONS];              // touch regions for the buttons
    var private touchY1[MAXBUTTONS];
    var private touchX2[MAXBUTTONS];
    var private touchY2[MAXBUTTONS];
    var private vButtonState[MAXBUTTONS];
    // button state flags (bit 0 = up:down state)
    var private vOldButtonState[MAXBUTTONS];
    // OLD button state flags (bit 0 = up:down state)

    var private lastkey;                             // last button pressed

    // messages
    var private touchState;                          // window touch status
    var private touchX;                              // window x position
    var private touchY;                              // window y position

    // local variables
    var n, x, y, x1, y1, x2, y2, oldFG, oldBG, oldFont, r, retval;

    // save the things we will change
    oldFont := peekW(TXT_FONT_ID);
    oldFG   := peekW(TEXT_COLOUR);
    oldBG   := peekW(TEXT_BACKGROUND);

    // functions methods
    gosub(subfunc), (Initialize, Update, Scan, RedrawTitleBar, RedrawStatusBar);

    // restore things we changed
    txt_FontID(oldFont);
    txt_FGcolour(oldFG);
    txt_BGcolour(oldBG);

    goto exitfunc;

    //---------------------------------------------------
    // reset the window and redraw the buttons to the up state
    //---------------------------------------------------
    Initialize:
        touch_Set(TOUCH_ENABLE);               // enable the touch screen
        gfx_Panel(PANEL_RAISED, xpos, ypos, windowWidth, titlebarheight,
titlebarcolor);                                  // draw title bar panel
        gfx_Panel(PANEL_SUNKEN, xpos, peekW(GFX_Y2), windowWidth, windowHeight-
titlebarheight-statusbarheight, windowcolor);  // draw main window panel
        gfx_Panel(PANEL_RAISED, xpos, peekW(GFX_Y2), windowWidth, statusbarheight,
statusbarcolor);              // draw status bar panel

        x := xpos+windowWidth-titlebarheight;
```

```
        y := ypos+2;
        gfx_Button(BUTTON_UP, x, y, OLIVE, ORANGE, FONT1, 1, 1,btnexit );
       // place the quit button

        gosub RedrawTitleBar;  // set the title
        gosub RedrawStatusBar; // set the status bar text

        x1 := xpos+10;
        y1 := ypos+30; // set position of the first button offset in the window

        for(n:=0; n<MAXBUTTONS-1; n++)    // draw the 4 ush buttons

            // reset the button states
            vButtonState[n]:=UP;
            vOldButtonState[n]:=UP;

            // place a button
            gfx_Button( BUTTON_UP, x1, y1, btncolor, textcolor, font, 1, 1,
buttontexts[n] );

            // get the bottom/right extent
            x2 := gfx_Get(RIGHT_POS);
            y2 := gfx_Get(BOTTOM_POS);

            // register the button position
            touchX1[n] := x1;
            touchY1[n] := y1;
            touchX2[n] := x2;
            touchY2[n] := y2;
            y1 := y2 + ygap;            // move down
        next

        touchX1[n] := x;              // finally, register exit button position
        touchY1[n] := y;
        touchX2[n] := x+20;
        touchY2[n] := y+20;
        vButtonState[n]:=UP;
        vOldButtonState[n]:=UP;

        title := "NO NAME";          // set default title bar text
        statusbartext := "status..."; // set default status bar text

        gosub RedrawStatusBar;
        gosub RedrawTitleBar;

        //Do any other initializions here

        endsub;

    //-------------------------------------------------
    // Update status bar text
    //-------------------------------------------------
     RedrawStatusBar:
        txt_FontID(FONT1);
        txt_FGcolour(statustextcolor);
        txt_BGcolour(statusbarcolor);
        gfx_MoveTo(xpos+8, ypos+windowHeight-8);
        putstr(statusbartext);       // set the status bar text
        endsub;

    //-------------------------------------------------
    // Update status bar text
    //-------------------------------------------------
     RedrawTitleBar:
        txt_FontID(FONT2);
        txt_FGcolour(titletextcolor);
        txt_BGcolour(titlebarcolor);
        gfx_MoveTo(xpos+8, ypos+5);
```

```
        putstr(title);                                      // set the title text
        endsub;

    //---------------------------------------------------
    // Update any buttons that have changed state
    //---------------------------------------------------
    Update:
         for(n:=0; n<MAXBUTTONS; n++)
            if ( vButtonState[n] != vOldButtonState[n])
                vOldButtonState[n] := vButtonState[n];
                gfx_Button( vButtonState[n], touchX1[n], touchY1[n], btncolor,
textcolor, font, 1, 1, buttontexts[n] ); // redraw the button

            endif
        next
        //Do any other update operations here....

    endsub;

    //---------------------------------------------------
    // scan for any button presses
    //---------------------------------------------------
    Scan:
    touchState := touch_Get(TOUCH_STATUS);  // save touchscreen status
    touchX := touch_Get(TOUCH_GETX);        // and current position
    touchY := touch_Get(TOUCH_GETY);

    // if screen touched,
    if(touchState == TOUCH_PRESSED)

        // scan the hot spots list
        while (n < MAXBUTTONS && !r)
            if (touchX >= touchX1[n] && touchX < touchX2[n] && touchY >= touchY1[n] &&
touchY < touchY2[n]) r := n+1;
            n++;
        wend

        // if any button was pressed
        if(r)
            lastkey := r;                           // remeber the button
            vButtonState[r-1] := DOWN;              // set it to down state
            gosub Update;                           // update the button action
        endif
    endif

    if((touchState == TOUCH_RELEASED) && lastkey)
    // if touch released and we remember a previous button press,
        retval := lastkey;              // set return value with the button number
        vButtonState[lastkey-1] := UP;// last button is now UP
        lastkey := 0;                   // clear button memory
        gosub Update;                   // update the button action
    endif
    endsub;

exitfunc:
    return retval;
endfunc

//=============================================================================
// mount the drive, return status message and D will be null if mount fails
//=============================================================================
func init_Drive()
    var retry := 10;
    if(!(D := file_Mount()))
        while(retry--)
            if((D := file_Mount())) break;
        wend
        if (retry) return "Mount Failed!";
```

```
    endif
    return "Disk mounted";
endfunc

//================================================================================
```

5. Appendix B : Runtime Error Messages

Error No.	Meaning	Category	
1	Failed to receive 'L' during loading process from Workshop	Workshop	
2	Did not receive valid header info from Workshop	Workshop	
3	Header size does not match loader info	Workshop	
4	Could not allocate enough memory for program	Workshop	
5	Loader checksum error	Workshop	
6	Did not receive header prior to 'L' command	Workshop	
7	Header size entry does not match loader value	Workshop	
8	Failed to load program from FLASH	Internal	
9	Could not allocate code segment	File loader	
10	Could not load function file from disk	File loader	
11	Bad header in program file	File loader	
12	Header in program file differs from file size	File loader	
13	Could not allocate global memory for program file	File loader	
14	Program File checksum error	File loader	
15	EVE Stack Overflow	System	
Error No.	**Meaning**	**V1**	**V2**
16	Unsupported PmmC function	fnc	1st Arg
17	Illegal COM0 Event Function address	addr	(ignored)
18	Illegal COM1 Event Function address	addr	(ignored)
19	Bad txt_Set(...) command number	command	value
20	Bad gfx_Get(...) command number	command	(ignored)
21	Bad txt_Set(...) command number	command	value
22	Bad address for peekW or pokeW	command	(ignored)
23	Bad timer number for sys_SetTimer(..) or sys_GetTimer(..)	tnum	value
24	Bad timer number for sys_SetTimerFunction(...)	tnum	funcaddr

6. Hardware Tools

The following hardware tools are required for full control of the Picaso Processor.

6.1. 4D Programming Tools

The 4D Programming Cable, uUSB-PA5-II and 4D-UPA Programming Adaptors, are essential hardware tools to program, customise and test the Picaso processor. Any one of these can be used.

The 4D programming interfaces are used to program a new Firmware/PmmC, Display Driver and for downloading compiled 4DGL code into the processor. They even serve as an interface for communicating serial data to the PC.

The 4D Programming Cable, uUSB-PA5-II Programming Adaptors, and 4D-UPA are available from 4D Systems, www.4dsystems.com.au

Using a non-4D programming interface could damage your processor, and **void your Warranty.**

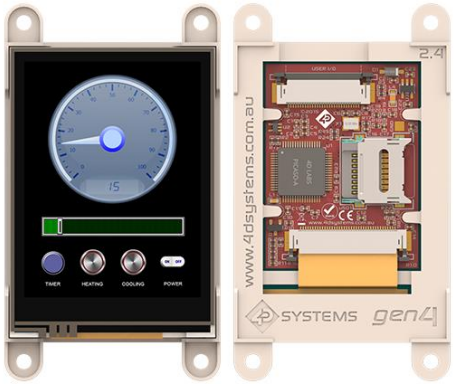
4D Programming Cable

uUSB-PA5-II Programming Adaptor

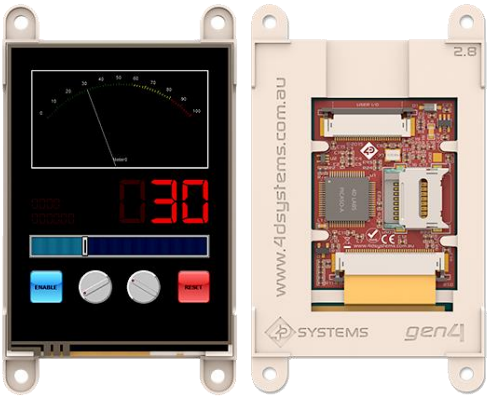
4D-UPA Programming Adaptor

6.2. Evaluation Display Modules

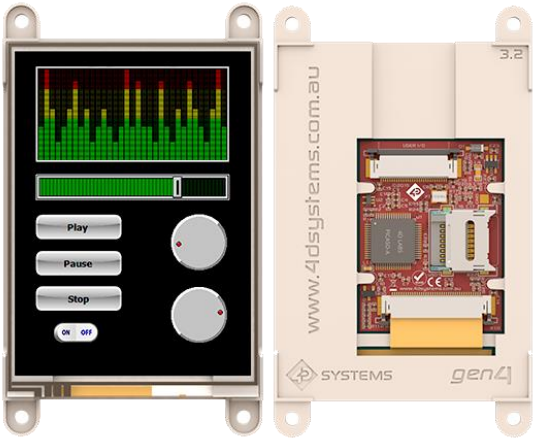
The following modules, available from 4D Systems, can be used for evaluation purposes to discover what the Picaso processor has to offer. Some of the range available are as follows:



gen4-uLCD-24PT – 2.4" Intelligent Picaso Display



gen4-uLCD-28PT – 2.8" Intelligent Picaso Display



gen4-uLCD-32PT – 3.2" Intelligent Picaso Display

NOTE: gen4 Picaso display module plastic ships in red colour, not white as pictured.

7. Workshop4 IDE

Workshop4 is a comprehensive software IDE that provides an integrated software development platform for all of the 4D family of processors and modules. The IDE combines the Editor, Compiler, Linker and Down- Loader to develop complete 4DGL application code. All user application code is developed within the Workshop4 IDE.

The Workshop4 IDE supports multiple development environments for the user, to cater for different user requirements and skill level.

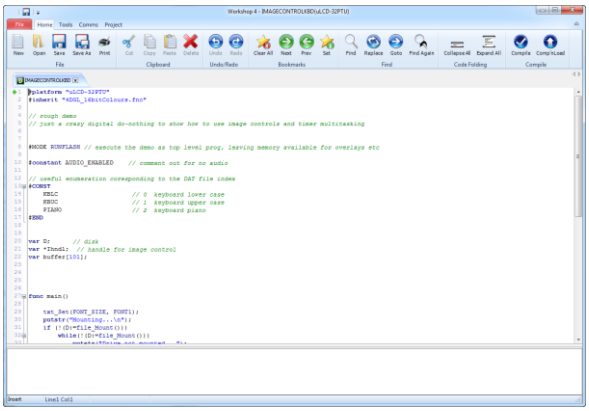
- The **Designer** environment enables the user to write 4DGL code in its natural form to program the Picaso module.
- A visual programming experience, suitably called **ViSi**, enables drag-and-drop type placement of objects to assist with 4DGL code generation and allows the user to visualise how the display will look while being developed.
- An advanced environment called **ViSi-Genie** doesn't require any 4DGL coding at all, it is all done automatically for you. Simply lay the display out with the objects you want, set the events to drive them and the code is written for you automatically. ViSi-Genie provides the latest rapid development experience from 4D Labs.
- A **Serial** environment is also provided to transform the Picaso module into a slave serial module, allowing the user to control the display from any host microcontroller or device with a serial port.

The Workshop4 IDE and comprehensive manuals and documentation is available from the 4D Systems website. www.4dsystems.com.au, specifically on the Workshop4 IDE product page.

7.1. Designer Environment

Choose the Designer environment to write 4DGL code in its raw form.

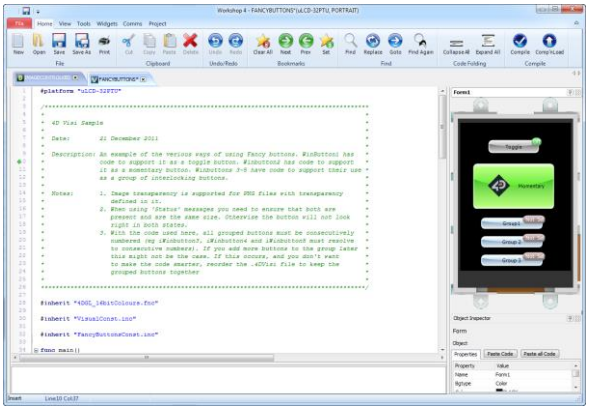
The Designer environment provides the user with a simple yet effective programming environment where pure 4DGL code can be written, compiled and downloaded to the Picaso.



7.2. ViSi Environment

ViSi was designed to make the creation of graphical displays a more visual experience.

ViSi is a great software tool that allows the user to see the instant results of their desired graphical layout. Additionally, there is a selection of inbuilt dials, gauges and meters that can simply be placed onto the simulated module display. From here each object can have its properties edited, and at the click of a button all relevant 4DGL code associated with that object is produced in the user program. The user can then write 4DGL code around these objects to utilise them in the way they choose.



7.3. ViSi Genie Environment

ViSi Genie is a breakthrough in the way 4D Labs' graphic modules are programmed.

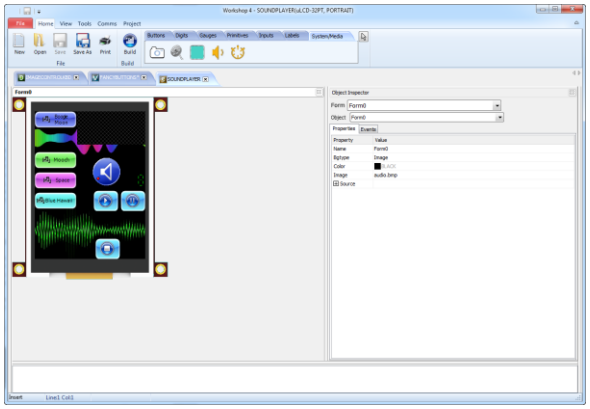
It is an environment like no other, a code-less programming environment that provides the user with a rapid visual experience, enabling a simple GUI application to be 'written' from scratch in literally seconds.

ViSi Genie does all the background coding, no 4DGL to learn, it does it all for you.

Pick and choose the relevant objects to place on the display, much like the ViSi Environment, yet without having to write a single line of code. Each object has parameters which can be set, and configurable events to animate and drive other objects or communicate with external devices.
Simply place an object on the screen, position and size it to suit, set the parameters such as colour, range, text, and finally select the event you wish the object to be associated with, it is that simple.

In seconds you can transform a blank display into a fully animated GUI with moving sliders, animated press and release buttons, and much more. All without writing a single line of code!

ViSi Genie provides the user with a feature rich rapid development environment, second to none.



7.4. Serial Environment

The Serial environment in the Workshop4 IDE provides the user the ability to transform the Picaso into a slave serial graphics controller.

This enables the user to use their favourite microcontroller or serial device as the Host, without having to learn 4DGL or program in a separate IDE. Once the Picaso is configured and downloaded to from the Serial Environment, simple graphic commands can be sent from the users host microcontroller to display primitives, images, sound or even video.

Refer to the *Picaso Serial Command Set Reference Manual* from the Workshop4 product page on the 4D Systems website for a complete listing of all the supported serial commands.

8. Revision History

Revision History		
Revision	**Revision Content**	**Revision Date**
1.0	First Release	20/06/2010
2.0	1-Incorrect heading and discrepancy in the description of bus_Write Function; fixed. 2-Fixed typing error in the bus_Read Function. 3-Erroneous references in sec 2.4.12 to “note #5”, “note #6”, “note #7” and “note #8” removed. Proper descriptions added. 4-Replaced FONT_SIZE with FONT_ID at several places. 5-X_RES is replaced with X_MAX. Y_RES is replaced with Y_MAX in sec 2.6.39 6-str_Append is replaced with str_Cat in the example in sec 2.16.16. 7-str_Append is replaced with str_CatN in the example in sec 2.16.17.	25/10/2010
3.0	1-Fixed typing error in Sec 2.19.1, Sec 2.19.2 and Sec 2.19.3. 2-Added Details for Transparency functions. Sec 2.6.41. 3-Added Details for uVGA-II/III related functions in Sec 2.6.41. 4-Sec 2.7.7 added. disp_Sync(line) command added for uVGA-II/III module. 5-Updated SPI modes and SPI speeds. Note SPI diagram in Sec 2.10.1 6-Fixed typing error in Sec 2.13.1 and 1.13.2. It's a 32 bit Timer. 7-Fixed typing error in the Description in Sec 2.12.1.	17/11/2011
4.0	1-Removed predefined numbers from table 2.6.41. gfxt_Set should only be used with predefined names. 2-Transparency, Contrast and Multiple Page Display/Read/Write details updated in Sec 2.6.41. 3-Fixed typo in 2.4.10 strheight(pointer). 4-Added Sec 2.3.11 CY(). 5-Added Sec 2.3.12 umul_1616(&res32, val1, val2) 6-Added Sec 2.3.13 uadd_3232(&res32, &val1, &val2) 7- Added Sec 2.3.14 usub_3232(&res32, &val1, &val2) 8-Added Sec 2.3.15 ucmp_3232(&val1, &val2) 9- Added Sec 2.16.18 str_ByteMove(src, dest, count) 10-Added Sec 2.16.19 str_Copy(dest, src) 11- Added Sec 2.16.20 str_CopyN(dest, src, count)	17/02/2012
5.0	1-Fixed typing error in the SWAP command. Sec 2.3.4 2-Fixed typing errors in Sec 2.4.10 3-Updated COM1 Default Baud rate details. 4-Fixed typing error in Example. sys_EventsPostpone in Sec 2.13.7 5-Added details to gfx_Cls() command Sec 2.6.1 6-com_TXbuffer and com1_TXbuffer functions have been modified and take an extra parameter. It applies to PmmC R29 or above. Sec 2.11.11 7-Description updated, Image control will now show error box for out of range video frames. If frame is set to -1, just a rectangle will be drawn in background colour to blank an image. It applies to PmmC R29 or above. Sec 2.18.6 8-Description updated, Image control will now show error box for out of range video frames. Also, if frame is set to -1, just a rectangle will be drawn in background colour to blank an image. It applies to PmmC R29 or above. Sec 2.8.13	08/06/2012
6.0	Reformatted, minor document updates	12/09/2012

Revision History Continued…		
Revision	**Revision Content**	**Revision Date**
6.1	Fixed minor TOC numbering issue 1-It is now possible for a parent to access child globals when using file_LoadFunction. Sec 2.14.25 updated. Example added. Applies to PmmC R31 and above. 2-sys_SetTimerEvent(timernum, function), description added. Sec 2.13.5. 3-Sec 2.11.12, com_TXbufferHold(state) added. 4-Sec 2.11.13, com_TXcount(), "Returns" part fixed. 5-com_TXemptyEvent(...) description updated. com_TXemptyEvent(Function) is changed to com_TXemptyEvent(FunctionAddress). Added a better example in Sec 2.11.14 .	23/11/2012
6.2	Fixed minor issues in the wording and return types of some functions File_ScreenCapture – Typo in x and y description Gfx_Origin – Incorrect description Mem_Free – Return was incorrect Gfx_Get – Some modes were not listed, these have been added File_Image – Return was incorrect Media_Flush – Return was incorrect Sys_Sleep – Note added File_Exists – Removed wildcard support description, this was not supported File_Run, File_Exec – Status should be Value, otherwise OK PutW, putC – Return was incorrect SetBaud – Some % Errors listed in the SetBaud table were incorrect - Updated	17/12/2012
6.3	Fixes to str_Length() example Fixes to typo in mem_AllocV name, and description of size type and return improved Fixes to mem_Alloc size type and return improved Addition to type of Size in mem_Set command, and addition on an example Fixes to sys_SetTimerEvent() example, fix of typo Improvements made to img_SetWord and img_GetWord constant listings Improvement of img_SetImageControl description	12/01/2013
6.4	Removed str_String from listing as it didn't exist Updated Section 7	01/02/2013
6.5	SCREEN_MODE constants fixed, incorrectly documented	04/02/2013
6.6	Addition content added to sys_SetTimerEvent description	07/02/2013
6.7	Corrections to Contrast values for EXCEPTIONS sections	13/02/2013
6.8	Touch Get explanation of Mode 1 and Mode 2 extended	17/02/2013
6.9	Updated I2C_Write returns	02/04/2013
6.10	Updated setbaud and com_SetBaud information	30/04/2013
6.11	Detaill added to descriptions of serout() and com_TXbuffer() functions	11/05/2013
6.12	gfx_Selection() function removed due to instability some time ago from the PmmC, however was left in this document in error	30/05/2013
6.13	Updated file_error() table, sentence in 2.4.5 updated, and comment in 2.13.3 updated	12/06/2013
6.14	Added detail to file_Write and file_Read functions regarding their pointers, removed incorrect information about uVGAII/III orientation, and gfx_TriangleFilled functions	05/07/2013

Revision History Continued…		
Revision	**Revision Content**	**Date**
6.15	Added missing disp_Init() function, and reworded the img_Touched() description	03/09/2013
6.16	Added new Functions disp_Disconnect() and sys_DeepSleep(). Fix spelling mistake in file_LoadImageControl	22/10/2013
6.17	Fixed error return codes in file_PlayWAV and added missing code. Removed uLCD-43PT option for SCREEN_RES	18/12/2013
6.18	Documented v4.0 PmmC's changes to files opened in append mode.	21/03/2014
6.19	Added information for file_LoadImageControl Mode 2. Updated control block size in file_Mount. Added information about source of uSD based font in txt_FontID. Added note about restriction of turning clipping on and off. Added information about the use of TRANSPARENCY. Clarified information about events.	22/12/2014
6.20	Added notes to comx_TXbufferHold. Improved return description for str_Match and str_MatchI. Added str_Printf to 'to' function. Updated example for str_Cat, str_CatN, str_Find, str_FindI, str_Match, str_MatchI and file_Exec.	14/07/2015
6.21	Added I_TOUCH_DISABLE to img_SetAttributes and img_ClearAttributes.	19/8/2016
7.0	Updated formatting and contents	01/05/2017
7.1	Updated formatting	21/03/2019

9. Legal Notice

Proprietary Information

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

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

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

Disclaimer of Warranties & Limitation of Liability

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

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

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

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

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

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

10. Contact Information

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