DIABLO-16 Serial (SPE) Command Set

Manual

Revision 2.4

Contents

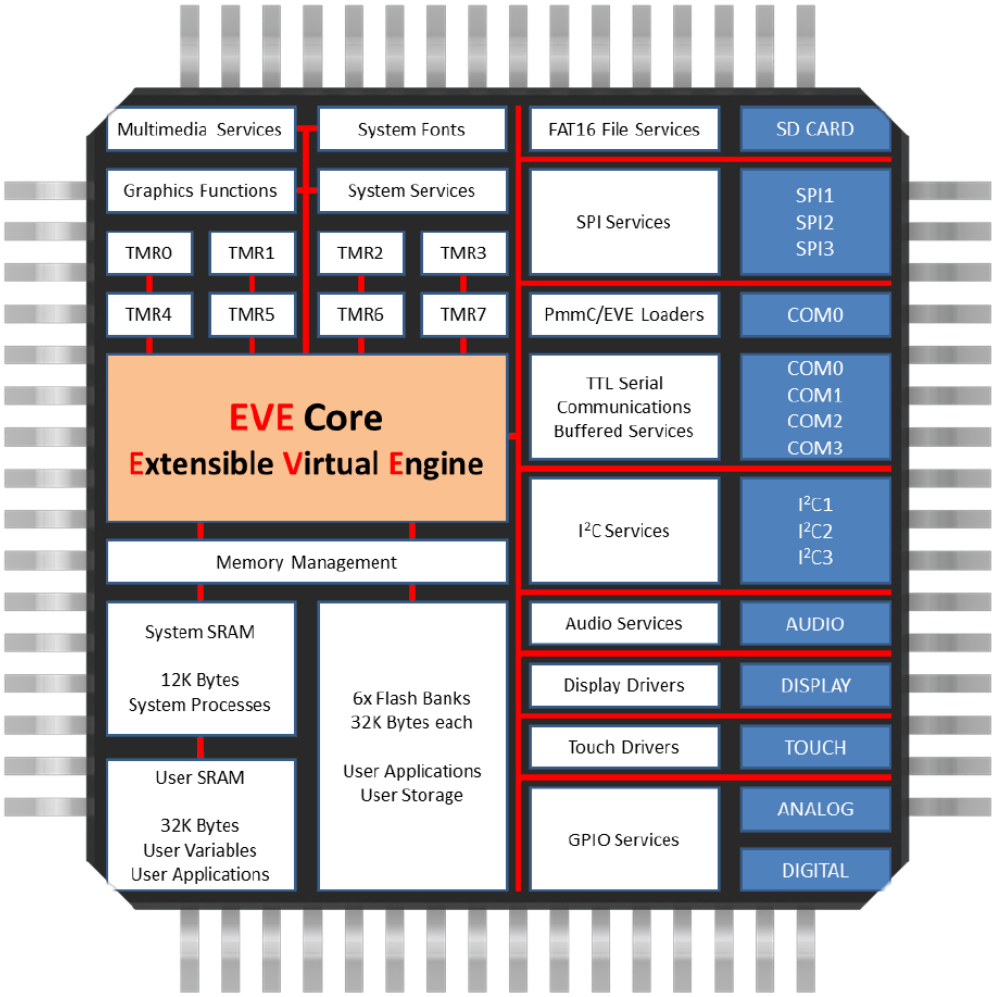
1. DIABLO-16 Processor

The 4D-Labs family of embedded graphics processors are powered by a highly optimised soft core virtual engine, E.V.E. (Extensible Virtual Engine).

There are many 4D Products powered with the DIABLO-16 processor, including:

- gen4-uLCD-35DT
- gen4-uLCD-43D/DT/DCT
- gen4-uLCD-70DT
- uLCD-220RD
- uLCD-70DT
- More coming soon...

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.



DIABLO-16 Internal Block Diagram

The DIABLO-16 processor used in the above products can be configured in a number of ways, depending on the needs of the user. Using the Workshop4 IDE by 4D Labs, the user has the choice of 4 programming environments, Designer, ViSi, ViSi-Genie and the Serial Environment.

This document targets the Serial Environment, how to configure a Display Module to be 'Serial Ready', and all the commands available in the Serial Environment to send the display from your Host Controller of choice.

For more information on the Workshop4 Software in General or the other Environments available in Workshop4, please refer to the Workshop4 User Guide, available from the 4D Labs website.

2. Introduction to Workshop4 - Serial Environment

The DIABLO-16 Processor can be programmed to act as a 'SERIAL SLAVE' device, responding to the Serial commands sent from virtually any Host Controller.

2.1. How to Configure Your Display Module as a Serial Slave

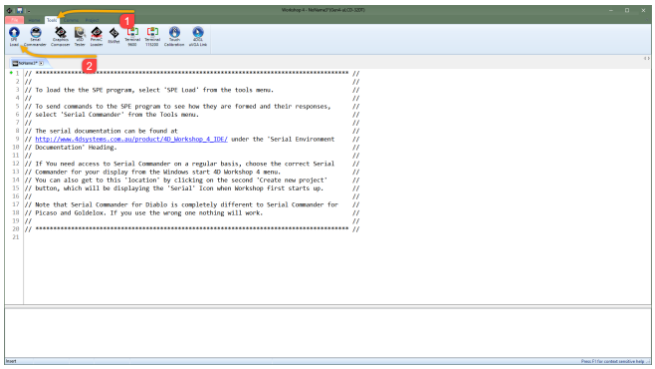
To set up your display module to be a Serial Display is a very simple process.

When a user starts the Workshop4 IDE, starts a new project, selects their module of choice, and then selects the Serial Environment, the user is presented with a basic environment to get them started using their chosen display as a Serial Slave.

Create New Project

Select a Display module

Select Serial Environment



Serial Environment

In the 'Tools' menu of the Serial Environment, is a button called 'SPE Load'. SPE stands for "Serial Platform Environment". If your display module is connected to the PC via the 4D Systems Programming Cable or Adaptor, clicking this button will load a special 4DGL application onto your module. This application is known as the SPE Application, and will enable your chosen module to run as a Serial Slave.

Note

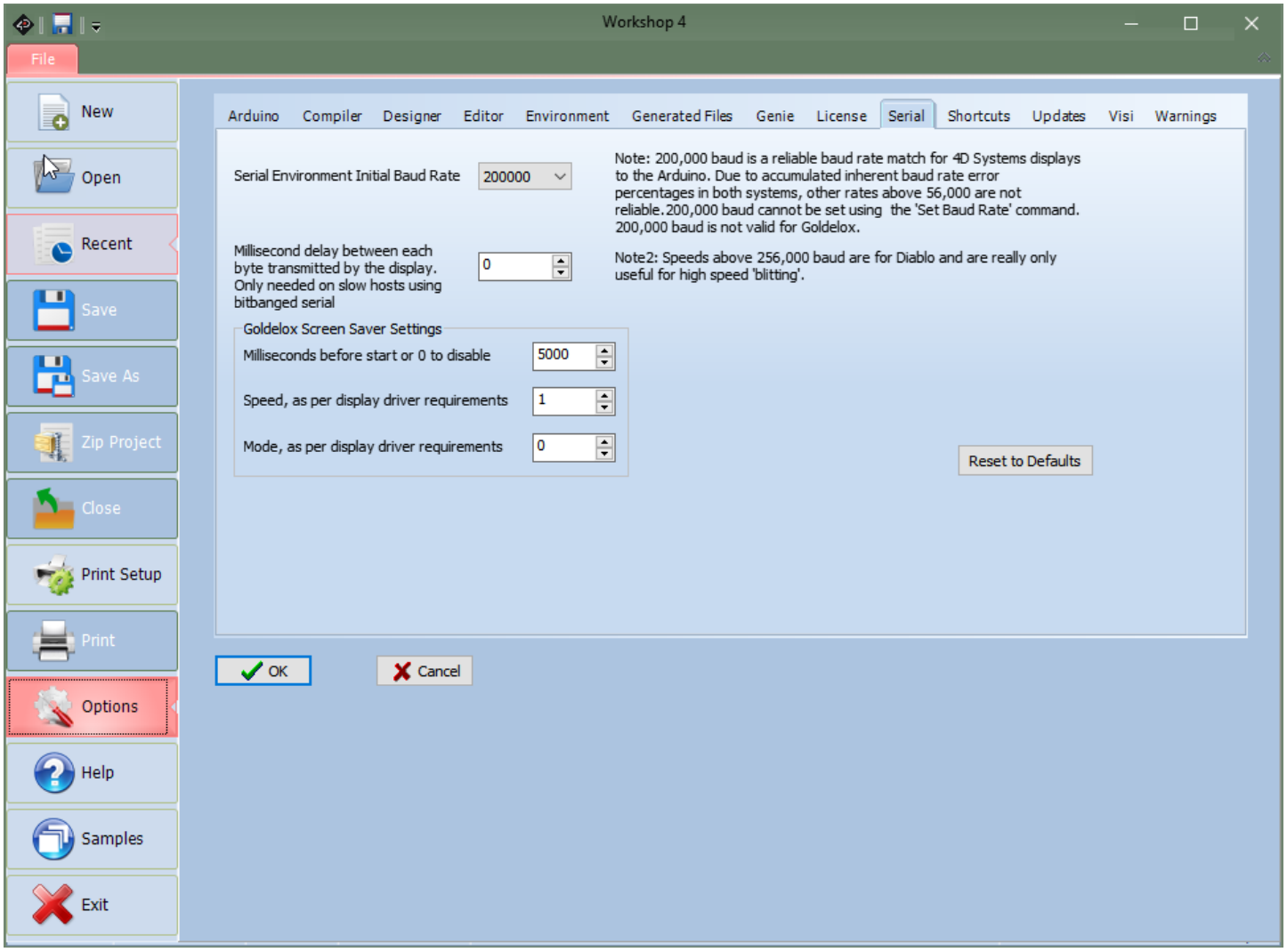
The Display Modules are **SPE READY** by default, meaning the SPE Application has been loaded to each of the modules at the 4D Systems Factory. The user can reload the **SPE** Application if required, to update the **SPE Application** on board OR to move over to the **Serial Environment** from another Workshop4 Environment such as Designer, ViSi or ViSi-Genie.

Once the chosen display module is 'SPE READY', either brand new out of the box, or programmed to have the SPE Application via the above instructions, the user can begin programming their Host of choice to communicate to the 4D Systems display module.

2.2. Additional Configuration Parameters for Serial Communication

When the SPE Application is loaded to the Display Module from the 4D Systems factory, the Baud Rate is set to the initial default of 9600. This initial Baud Rate can be modified, so when the Display Module starts up, it is at the desired Baud Rate without having to send commands to change it from the Host.

To change the default Baud Rate, click on the Option button on the buttons down the left hand side of the Workshop4 IDE, click on the Serial tab, and change the 'Serial Environment Initial Baud Rate' to be whatever is suitable for your application.



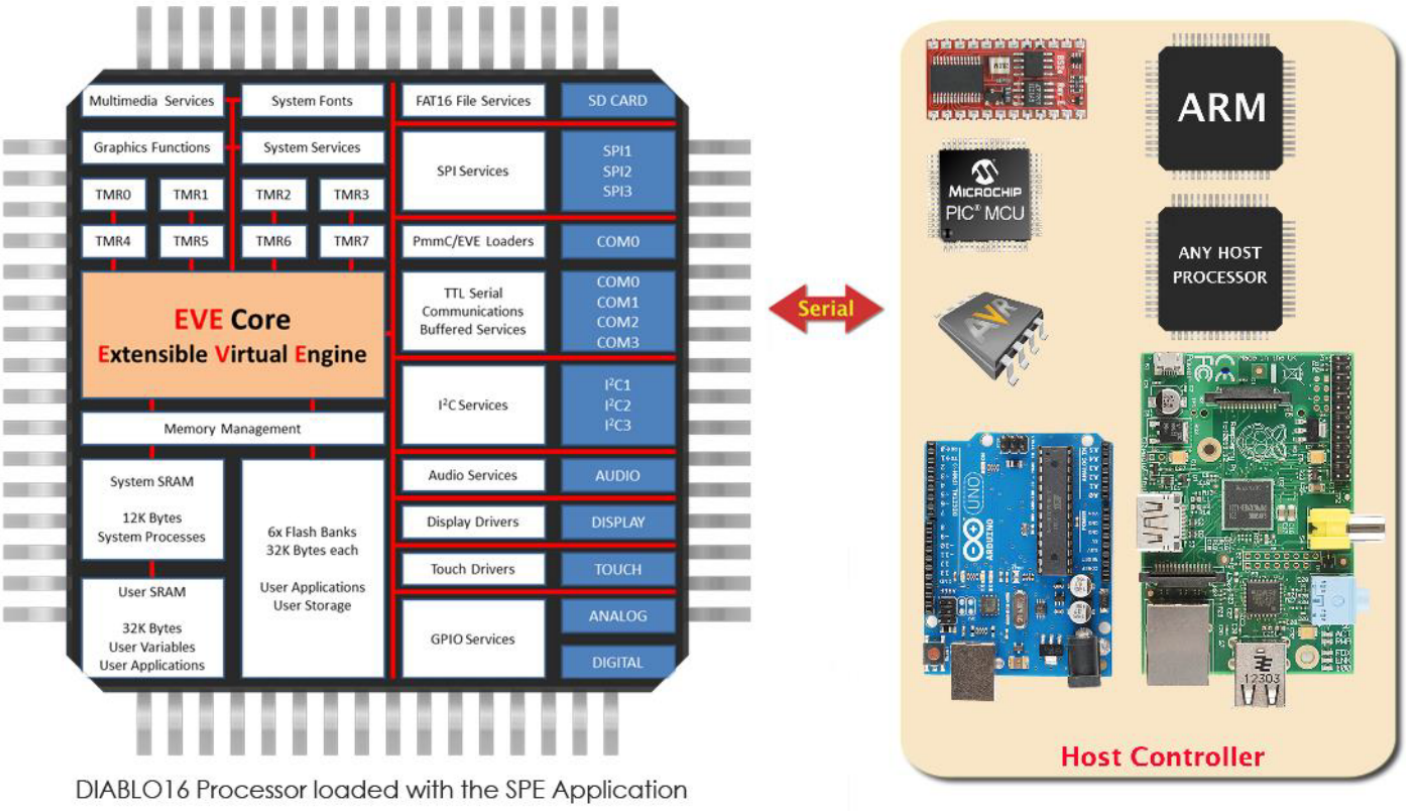
> **Note**
>
> The initial Baud rate and 'slowdown' settings for slow systems can be set under 'options', 'serial' before loading SPE.

Once the desired Baud Rate has been set, along with any 'Slowdown' delay (where required), the Display Module needs to have the SPE Application loaded once again, so these settings can take effect.

Simply follow the instructions in the How to Configure Your Display Module as a Serial Slave section, to load the updated SPE Application onto the Display Module.

2.3. Host Interface

When a Display Module is loaded with the SPE Application, it enables communication to a Serial Host over a bidirectional serial interface via one of its Serial UART's. All communications between the host and the device occur over these serial interfaces. The protocol is simple and easy to implement.



Serial Data Format: 8 Bits, No Parity, 1 Stop Bit. Serial data is true and not inverted

2.4. Introduction and Guidelines to the Serial Protocol

The Serial Protocol used with the SPE Application is a set of commands with associated parameters, to enable the Host Controller to display primitives, text, images, play audio, video or data log to micro-SD card, receive touch events etc on the 4D Systems Display Module, in the simplest manner available.

The Serial Protocol is made up of commands and parameters, sent over the Serial Port in byte format to the Display Module. Each command is unique and has a specific set of parameters associated with it. Each command that is sent to the Display Module is replied to with a response. Some commands do not specifically require a response, so for these commands the Display will reply with an Acknowledgment once successfully executed.

Commands that require a specific response may send back a varying number of bytes, depending on the command and what the response is.

Each Command sent to the display will require a certain amount of time before the response is sent, again dependent on the command and the operation that has to be performed.

Commands should only be sent and their response received, before another command is sent. If two commands are sent before the first response is received, incorrect operation may follow.

2.5. Power-Up and Reset

When the DIABLO-16 processor comes out of a power-up or external reset, a sequence of events is executed internally. The user should wait at least 3 seconds for the start-up to take place before attempting to communicate with the module.

2.6. Splash Screen

The splash screen appears on the screen 5 seconds after the start-up routines have been executed, provided there has been no serial activity.

The Splash screen can be customised if required. Please contact the 4D Systems Support team for more information on how this is done. This can be useful when integrating a 4D Systems product into a custom product, and SPE will be used, so it can be customised for your company/product requirements.

2.7. Power Supply

When powering DIABLO-16-powered displays, odd behaviour can be experienced if it is not supplied sufficient current. This is especially noticeable when powering the Host Controller board and the Display Module from the same USB port of your computer.

Please ensure you power your DIABLO-16-powered displays from a suitable power supply, based on the requirements of the display module, specified in the individual datasheets.

3. The Serial Command Set - Explained

The Serial Protocol and associated Commands enable the user to send bytes serially from the chosen Host Controller, to the 4D Display module loaded with the SPE Application, and control or receive information from, the Display Module.

In the DIABLO-16 Serial Protocol Command Set, there are currently 143 Commands available to the user. Each command send to the Display Module will incur a response of some description from the Display Module. This may be in the form of data, or a simple ACK that the command has been received.

Here is an example to better illustrate a few commands.

3.1. Example 1 – Moving the Cursor

Aim: Moving the Cursor to a specific location on the display, so text can originate from that point.

MoveCursor Command: HEX 0xFFF0 (2 bytes) – (Library Function txt_MoveCursor)
MoveCursor Parameters: Line Number (2 bytes), Row Number (2 bytes)
MoveCursor Returns: Acknowledge HEX 0x06

To Move the Cursor to Line Number=7, Row Number=12, firstly the 7 and 12 need to be converted into bytes. 7 is 0x7 and 12 is 0x0C. Because the command requires 2 bytes for each of these parameters to be sent, the first byte in this example will be 0x00 for both the Line and the Row.

The Bytes that will need to be sent will be: **0xFF, 0xF0, 0x00, 0x07, 0x00, 0x0C**
The Bytes that will be received back from the display will be: **0x06**

3.2. Example 2 – Drawing a Hollow Rectangle

Aim: Draw a Hollow Rectangle at a specific location on the display, with a specific outline colour

Rectangle Command: HEX 0xFF7A (2 bytes) – (Library Function gfx_Rectangle)
Rectangle Parameters: X1 Position (2 bytes), Y1 Position (2 bytes), X2 Position (2 bytes), Y2 Position (2 bytes), Colour (2 bytes)
Rectangle Returns: Acknowledge HEX 0x06

To draw a Blue rectangle starting with the top left corner at X=100, Y=100 and the bottom right corner at X=200, Y=250, firstly the 100, 200 and 250 numbers need to be converted into bytes.

100 is 0x64, 200 is 0xC8 and 300 is 0x012C. Because the command requires 2 bytes for each of these parameters to be sent, the first byte in this example will be 0x00 for X1, Y1, and X2. Y2 utilises 2 bytes.

Finally, the colour needs to be sent as 2 bytes. The colour Blue is 0x001F.

The Bytes to be sent will be: **0xFF, 0x7A, 0x00, 0x64, 0x00, 0x64, 0x00, 0xC8, 0x01, 0x2C, 0x00, 0x1F**
The Bytes that will be received back from the display will be: **0x06**

Note

Separation commas ',' between bytes that are shown in the Bytes to Send, and the Bytes Received syntax are purely for legibility purposes in this document and must not be considered as part of any transmitted/received data unless specifically stated.

4. Using Serial with a Library

4.1. Available Libraries

4D Labs has created a set of libraries suitable for a range of microcontrollers on the market to use and communicate with 4D Labs' range of processors, when configured to be Serial Slaves using the SPE application and the Serial Environment in Workshop4.

The following libraries have been created and are **available from the Samples menu inside the Workshop4 IDE Software**, where the Workshop4 software is available from the 4D Labs website.

- Arduino Library
- C Library
- Pascal Library
- PicAxe Library

These libraries enable the programmer to have access to all of the Serial Commands, but in a format that is more suited for High Level Programming, such as the Arduino IDE.

4.2. Benefits to using a Library

The libraries created by 4D Labs enable the user to simply include the library file in the code of their chosen Host Controller, and call high level functions (very similar and often equivalent to the 4DGL set of functions) instead of having to deal with the low level serial data bytes.

Please refer to the individual application notes on each of the libraries (as they become available), for a better understanding of what they include and how they are used in a Host controller. Refer to the Workshop4 product page on the 4D Labs website for more information, along with the modules product page.

4.3. Basic Example of using a library

If using the Arduino as the host controller of choice, by simply copying the library into the appropriate libraries folder for the Arduino IDE, and including the library in your sketch, the Arduino user will then have access to high level functions which provide many benefits over using the low level byte commands.

For example, to clear the display, and draw a rectangle from X1=10, Y1=110 to X2=200, Y2=220 in Red on the display, the following byte commands are required:

> **Send to the display:** 0xFF, 0x82 **Receive from the display**: 0x06 **Send to the display:** 0xFF, 0x7A, 0x00, 0x0A, 0x00, 0x6E, 0x00, 0xC8, 0x00, 0xDC, 0xF8, 0x00 **Receive from the display:** 0x06

Sending these commands from the Arduino would require each byte to be sent over the serial port to the display. 4D Labs has created a library to do this for you.

Using the Arduino library for example, the following functions would be required:

```
Display.gfx_Cls();
Display.gfx_Rectangle(10, 110, 200, 220, RED);
```

4.4. Library References

While this document is specifically for the Serial Command bytes, at the bottom of each command table is a reference to the relevant function that would be called if using the 4D Labs Serial Library.

5. DIABLO-16 Serial Commands

The following sections detail each of the commands available in the 4D Labs Serial Environment, when communicating to a 4D Systems Display Module loaded with the SPE Application. Please refer to the How to Configure Your Display Module as a Serial Slave section for more information on how to do this.

DIABLO-16 Serial commands can be categorized based on usage as listed below:

- FAT16 File Commands
- Graphics Commands
- I/O Commands
- Image Control Commands
- Media Commands (SD/SDHC Memory cards)
- Serial (UART) Communications Commands
- Sound Control Commands
- System Commands
- Text and String Commands
- Timer Commands
- Touch Screen Commands

5.1. Text and String Commands

The following is a summary of the commands available to be used for Text and Strings:

- Move Cursor
- Put Character
- Put String
- Character Width
- Character Height
- Text Foreground Colour
- Text Background Colour
- Set Fonts
- Text Width
- Text Height
- Text X-Gap
- Text Y-Gap
- Text Bold
- Text Inverse
- Text Italic
- Text Opacity
- Text Underline
- Text Attributes
- Text Wrap

5.1.1. Move Cursor

The **Move Cursor** command 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 "Move Origin" command 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.

Library Function: `txt_MoveCursor`

Syntax: `cmd(word)`, `line(word)`, `column(word)`

Commands	Description
cmd	0xFFF0
line	Holds a positive value for the required line position.
column	Holds a positive value for the required column position.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), line(MSB), line(LSB), column(MSB), column(LSB)

0xFF, 0xF0, 0x00, 0x05, 0x00, 0x03

// This will move the cursor to Line=5, Column=3
// Where 5 as 2 byes is 0x00 and 0x05, and 3 as 2 bytes is 0x00 and 0x03

// The Response will be 0x06 if the command is successfully executed
```

See also

See also the "Move Origin" command in the Graphics Commands section to move the origin to an exact pixel on the screen, which is suitable for both text and graphics.

5.1.2. Put Character

The **Put Character** command prints a single character to the display.

Library Function: `putCH`

Syntax: `cmd(word)`, `character(word)`

Commands	Description
cmd	0xFFFE
character	Holds a positive value for the required character.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), character(MSB), character(LSB)

0xFF, 0xFE, 0x00, 0x39

// This will send the character '9' (0x00, 0x39) to the display

// The Response will be 0x06 if the command is successfully executed
```

See also

See also the "Move Origin" command in the Graphics Commands section to move the origin to an exact pixel on the screen, which is suitable for both text and graphics.

5.1.3. Put String

The **Put String** command prints a string to the display. The argument can be a string constant or a pointer to a string.

A string needs to be terminated with a NULL.

Library Function: `putstr`

Syntax: `cmd(word)`, `string(string)`

Commands	Description
cmd	0x0018
string	Holds a Null terminated string char0, char1, char2, ..., charN, NULL

Note

Maximum characters in the string is 511 + NULL

Returns: `acknowledge(byte)`, `stringlength(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *stringlength*: Length of the string printed

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), char0, char1, char2, ..., charN, NULL

0x00, 0x18, 0x48, 0x65, 0x6C, 0x6C, 0x6F, 0x00

// This will send the string “Hello” to the display, as H = 0x48, e = 0x65, l = 0x6C
// and o = 0x6F, followed by a NULL = 0x00.

// The response will be 0x06, 0x00, 0x05 indicating ACK followed by the number 5 for
// length expressed as 2 bytes (1 word).
```

See also

See also the “Move Origin” command in the Graphics Commands section to move the origin to an exact pixel on the screen, which is suitable for both text and graphics.

5.1.4. Character Width

The **Character Width** command is used to calculate the width in pixel units for a character, based on the currently selected font. The font can be proportional or monospaced. If the total width of the character exceeds 255 pixel units, the function will return the 'wrapped' (modulo 8) value.

Library Function: `charwidth`

Syntax: `cmd(word)`, `char(byte)`

Commands	Description
cmd	0x001E
char	The ASCII character for the width calculation.

Returns: `acknowledge(byte)`, `width(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *width*: Width of a single character in pixel units.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), char

0x00, 0x1E, 0x65

// This is requesting the width in pixels of the character ‘e’, as ASCII ‘e’ is Hex 0x65

// Assuming for example the selected font is FONT_3

// The response will be 0x06, 0x00, 0x08 where 0x00, 0x08 is Decimal 8
// (FONT_3 is a 12x8 font)
```

5.1.5. Character Height

The **Character Height** command is used to calculate the height in pixel units for a character, based on the currently selected font. The font can be proportional or monospaced. If the total height of the character exceeds 255 pixel units, the function will return the 'wrapped' (modulo 8) value.

Library Function: `charheight`

Syntax: `cmd(word)`, `char(byte)`

Commands	Description
cmd	0x001D
char	The ASCII character for the height calculation.

Returns: `acknowledge(byte)`, `height(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *width*: Height of a single character in pixel units.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), char

0x00, 0x1D, 0x65

// This is requesting the height in pixels of the character ‘e’, as ASCII ‘e’ is Hex 0x65

// Assuming for example the selected font is FONT_3

// The response will be 0x06, 0x00, 0x0C where 0x00, 0x0C is Decimal 12
// (FONT_3 is a 12x8 font)
```

5.1.6. Text Foreground Colour

The **Text Foreground Colour** command sets the text foreground colour, and reports back the previous foreground colour.

Library Function: `txt_FGcolour`

Syntax: `cmd(word)`, `colour(word)`

Commands	Description
cmd	0xFFEE
colour	Specifies the colour to be set.

Returns: `acknowledge(byte)`, `colour(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *colour*: Previous Text Foreground Colour.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), colour(MSB), colour(LSB)

0xFF, 0xEE, 0x00, 0x10

// This is setting the Foreground colour to Navy, which is Hex 0x00, 0x10

// The Response will be 0x06, 0x04, 0x00 assuming the previous colour was Green,
// which is 0x04, 0x00
```

5.1.7. Text Background Colour

The **Text Background Colour** command sets the text background colour, and reports back the previous background colour.

Library Function: `txt_BGcolour`

Syntax: `cmd(word)`, `colour(word)`

Commands	Description
cmd	0xFFED
colour	Specifies the colour to be set.

Returns: `acknowledge(byte)`, `colour(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *colour*: Previous Text Background Colour.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), colour(MSB), colour(LSB)

0xFF, 0xED, 0xF8, 0x00

// This is setting the Background colour to Red, which is Hex 0xF8, 0x00

// The Response will be 0x06, 0x00, 0x10 assuming the previous colour was Navy,
// which is 0x00, 0x10
```

5.1.8. Set Font

The **Set Font** command sets the required font using its ID, and report back the previous Font ID used.

Library Function: `txt_FontID`

Syntax: `cmd(word)`, `id(word)`

Commands	Description
cmd	0xFFEC
id	1 for FONT_1 = System 5x7 2 for FONT_2 = System 8x8 3 for FONT_3 = System 8x12 (Default) 4 for FONT_4 = System 12x16 5 for FONT_5 = MS San Serif 8x12 6 for FONT_6 = Deja Vu Sans 9pt 7 for FONT_7 = Deja Vu Sans Bold 9pt 8 for FONT_8 = Deja Vu Sans Condensed 9pt 9 for FONT_9 = System 3x6 10 – Not currently available for SPE Serial, N/A 11 for FONT_11 = EGA 8x12 font

Note

The value could also be the handle of a uSD based font obtained using file_LoadImageControl(). The font would generally have been generated using a Strings object in ViSi (easy) or from a the FONT TOOL (harder). (Please refer to the Application Notes).

Preferably use the FONT_1 through FONT_11 predefined constants.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Font ID.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), id(MSB), id(LSB)

0xFF, 0xEC, 0x00, 0x02

// This will set the font to be FONT_2 which is 0x00, 0x02

// The response will be 0x06, 0x00, 0x01 assuming the previous font was FONT_1,
// where FONT_1 is 0x00, 0x01
```

5.1.9. Text Width

The **Text Width** command sets the text width multiplier between 1 and 16, and returns the previous multiplier.

Library Function: `txt_Width`

Syntax: `cmd(word)`, `multiplier(word)`

Commands	Description
cmd	0xFFEB
multiplier	Width multiplier 1 to 16 (Default =1)

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Multiplier value.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), multiplier(MSB), multiplier(LSB)

0xFF, 0xEB, 0x00, 0x05

// This will set the Text Width to be 5x that of the default

// The response will be 0x06, 0x00, 0x01 assuming the previous Text width multiplier
// was 1 (0x00, 0x01)
```

5.1.10. Text Height

The **Text Height** command sets the text height multiplier between 1 and 16, and returns the previous multiplier.

Library Function: `txt_Height`

Syntax: `cmd(word)`, `multiplier(word)`

Commands	Description
cmd	0xFFEA
multiplier	Height multiplier 1 to 16 (Default =1)

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Multiplier value.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), multiplier(MSB), multiplier(LSB)

0xFF, 0xEA, 0x00, 0x02

// This will set the Text Height to be 2x that of the default

// The response will be 0x06, 0x00, 0x01 assuming the previous Text height multiplier
// was 1 (0x00, 0x01)
```

5.1.11. Text X-gap

The **Text X-gap** command sets the pixel gap between characters (x-axis), where the gap is in pixel units, and the response is the previous pixelcount value.

Library Function: `txt_Xgap`

Syntax: `cmd(word)`, `pixelcount(word)`

Commands	Description
cmd	0xFFE9
pixelcount	0 to 32(Default =0)

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous pixelcount value.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), pixelcount(MSB), pixelcount(LSB)

0xFF, 0xE9, 0x00, 0x02

// This will set the text X-Gap to be 2 pixels, where 2 pixels is 0x00, 0x02

// The response will be 0x06, 0x00, 0x00 assuming the previous text X-gap was 0
```

5.1.12. Text Y-gap

The **Text Y-gap** command sets the pixel gap between characters (y-axis), where the gap is in pixel units, and the response is the previous pixelcount value.

This command is required to be used if setting text to have an 'Underline' using the "Text Underline" command, or "Text Attributes" command with the suitable bits set. See these command for further information.

Library Function: `txt_Ygap`

Syntax: `cmd(word)`, `pixelcount(word)`

Commands	Description
cmd	0xFFE8
pixelcount	0 to 32(Default =0)

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous pixelcount value.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), pixelcount(MSB), pixelcount(LSB)

0xFF, 0xE8, 0x00, 0x05

// This will set the text Y-Gap to be 5 pixels, where 5 pixels is 0x00, 0x05

// The response will be 0x06, 0x00, 0x00 assuming the previous text Y-gap was 0
```

5.1.13. Text Bold

The **Text Bold** command sets the Bold attribute for the text and report back the previous bold status.

Library Function: `txt_Bold`

Syntax: `cmd(word)`, `mode(word)`

Commands	Description
cmd	0xFFE5
mode	1 for ON. 0 for OFF.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Bold status.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB)

0xFF, 0xE5, 0x00, 0x01

// This will set the text to be bold, Bold = ON

// The response will be 0x06, 0x00, 0x00 assuming the previous bold status was OFF
// which is 0x00, 0x00
```

5.1.14. Text Inverse

The **Text Inverse** command sets the text to be inverse, and return the previous inverse status.

Library Function: `txt_Inverse`

Syntax: `cmd(word)`, `mode(word)`

Commands	Description
cmd	0xFFE3
mode	1 for ON. 0 for OFF.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous 'Text Inverse' status.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB)

0xFF, 0xE3, 0x00, 0x01

// This will set the text to be inverse, where inverse = ON = 0x00, 0x01

// The response will be 0x06, 0x00, 0x00 assuming the previous inverse status was OFF,
// which is 0x00, 0x00
```

5.1.15. Text Italic

The **Text Italic** command sets the text to italic, and return the previous text italic status.

Library Function: `txt_Italic`

Syntax: `cmd(word)`, `mode(word)`

Commands	Description
cmd	0xFFE4
mode	1 for ON. 0 for OFF.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Italic Text status.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB)

0xFF, 0xE4, 0x00, 0x01

// This will set the text to be italic, where italic = ON = 0x00, 0x01

// The response will be 0x06, 0x00, 0x00 assuming the previous inverse status was OFF,
// which is 0x00, 0x00
```

5.1.16. Text Opacity

The **Text Opacity** command selects whether the 'background' pixels are drawn, and returns the previous text opacity status. (Default mode is OPAQUE with BLACK background).

Library Function: `txt_Opacity`

Syntax: `cmd(word)`, `mode(word)`

Commands	Description
cmd	0xFFE6
mode	1 for ON. (Opaque) 0 for OFF. (Transparent)

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Text Opacity status.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB)

0xFF, 0xE6, 0x00, 0x00

// This will set the text to be transparent, where Opacity = OFF = 0x00, 0x00

// The response will be 0x06, 0x00, 0x01 assuming the previous opacity status was ON,
// which is 0x00, 0x01
```

5.1.17. Text Underline

The **Text Underline** command sets the text to underlined, and return the previous text underline status.

> **Note**
>
> The "Text Y-gap" command is required to be at least 2 for the underline to be visible, please refer to the "Text Y-gap" command for further information.

Library Function: `txt_Underline`

Syntax: `cmd(word)`, `mode(word)`

Commands	Description
cmd	0xFFE2
mode	1 for ON 0 for OFF.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Text Underline status.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB)

0xFF, 0xE2, 0x00, 0x01

// This will set the text to be underlined, where Underline = ON = 0x00, 0x01

// The response will be 0x06, 0x00, 0x00 assuming the previous underline status was OFF,
// which is 0x00, 0x00
```

5.1.18. Text Attributes

The Text Attributes command controls the following functions grouped:

- Text Bold
- Text Italic
- Text Inverse
- Text Underlined

Returns the previous Text Attributes status

> **Note**
>
> The "Text Y-gap" command is required to be at least 2 for the underline (Text Underlined attribute) to be visible, please refer to the "Text Y-gap" command for further information.

Library Function: `txt_Attributes`

Syntax: `cmd(word)`, `value(word)`

Commands	Description
cmd	0xFFE1
mode	(bit 5 or) DEC 16 for BOLD (bit 6 or) DEC 32 for ITALIC (bit 7 or) DEC 64 for INVERSE (bit 8 or) DEC 128 for UNDERLINED Set or Clear the relevant bits to set the attributes for the text to be written. (bits can be combined by using logical 'OR' of bits) **NOTE**: bits 0-3 and 8-15 are reserved

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Text Attributes status.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), value(MSB), value(LSB)

0xFF, 0xE1, 0x00, 0x90

// This will set the Text Attributes to be Bold and Underlined. Where Bold has the value 16
// and Underlined has the value 128, so 16+128=144 which is 0x90 in Hex.

// The response will be 0x06, 0x00, 0x00 assuming the previous attributes were No Bold,
// No Italic, No Inverse and No Underline.
```

5.1.19. Text Wrap

The **Text Wrap** command 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.

Library Function: `txt_Wrap`

Syntax: `cmd(word)`, `value(word)`

Commands	Description
cmd	0xFFE0
value	0 for OFF. 1 to N for ON, in Pixels.

Returns: `acknowledge(byte)`, `previous(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *previous*: Returns the previous wrap position.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB)

0xFF, 0x0E, 0x01, 0xA4

// This will set the wrap position to be at Pixel 420 from the left of the display,
// where Wrap = ON at pixel 420 = 0x01, 0xA4

// The response will be 0x06, 0x00, 0x00 assuming the previous wrap position was OFF,
// which is 0x00, 0x00
```

5.2. Graphics Commands

The following is a summary of the commands available to be used for Graphics:

- Clear Screen
- Change Colour
- Draw Circle
- Draw Filled Circle
- Draw Line
- Draw Rectangle
- Draw Filled Rectangle
- Draw Polyline
- Draw Polygon
- Draw Filled Polygon
- Draw Triangle
- Draw Filled Triangle
- Calculate Orbit
- Put Pixel
- Read Pixel
- Move Origin
- Draw Line and Move Origin
- Clipping
- Set Clip Window
- Extend Clip Region
- Draw Ellipse
- Draw Filled Ellipse
- Draw Button
- Draw Panel
- Draw Slider
- Screen Copy Paste
- Bevel Shadow

- Bevel Width
- Background Colour
- Outline Colour
- Contrast
- Frame Delay
- Line Pattern
- Screen Mode
- Transparency
- Transparent Colour
- Set Graphics Parameters
- Get Graphics Parameters

5.2.1. Clear Screen

The **Clear Screen** command clears the screen using the current background colour. This 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.

Library Function: `gfx_Cls`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFF82

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFF, 0x82

// The following will clear the display and restore the settings back to their defaults.

// The response will be 0x06 if the command is successful
```

5.2.2. Change Colour

The **Change Colour** command changes all **oldColour** pixels to **newColour** within the clipping window area.

Library Function: `gfx_ChangeColour`

Syntax: `cmd(word)`, `oldColour(word)`, `newColour(word)`

Commands	Description
cmd	0xFF69
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.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), oldColour(MSB), oldColour (LSB), newColour(MSB), newColour (LSB)

0xFF, 0x69, 0x00, 0x00, 0x00, 0x1F

// This will change all pixels coloured Black (0x00, 0x00) to be coloured Blue (0x00, 0x1F)
// within the clipping area. (Refer to the Clip Window command for more information on
// this.)

// The Response will be 0x06 if the command is successful
```

5.2.3. Draw Circle

The **Draw Circle** command draws a circle with centre point x, y with radius r using the specified colour.

Library Function: `gfx_Circle`

Syntax: `cmd(word)` , `x(word)` , `y(word)` , `rad(word)` , `colour(word)`

Commands	Description
cmd	0xFF78
x, y	Specifies the centre of the circle
rad	Specifies the radius of the circle
colour	Specifies the colour of the circle

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x(MSB), x(LSB), y(MSB), y(LSB), rad(MSB), rad(LSB), colour(MSB),
// colour(LSB)

0xFF, 0x78, 0x00, 0x64, 0x01, 0x2C, 0x00, 0x14, 0x80, 0x10

// This will draw a Circle at X=100 (Hex 0x00, 0x64), Y=300 (Hex 0x01, 0x2C),
// of Radius=20 (0x00, 0x14), and of Colour=Purple (0x80, 0x10).

// The response will be 0x06 if the command is successful
```

5.2.4. Draw Filled Circle

The **Draw Filled Circle** command draws a solid circle with centre point x1, y1 with radius using the specified colour. The outline colour can be specified with the "Outline Colour" command. If "Outline Colour" is set to 0, no outline is drawn.

Library Function: `gfx_CircleFilled`

Syntax: `cmd(word)` , `x(word)` , `y(word)` , `rad(word)` , `colour(word)`

Commands	Description
cmd	0xFF77
x, y	Specifies the centre of the circle
rad	Specifies the radius of the circle
colour	Specifies the colour of the circle

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x(MSB), x(LSB), y(MSB), y(LSB), rad(MSB), rad(LSB), colour(MSB),
// colour(LSB)

0xFF, 0x77, 0x00, 0x96, 0x00, 0xE6, 0x00, 0x32, 0x84, 0x10

// This will draw a Solid Filled Circle at X=150 (Hex 0x00, 0x96), Y=230 (Hex 0x00, 0xE6),
// of Radius=50 (0x00, 0x32), and of Colour=Grey (0x84, 0x10).

// The response will be 0x06 if the command is successful
```

5.2.5. Draw Line

The **Draw Line** command 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 "Line Pattern" command.

Library Function: `gfx_Line`

Syntax: `cmd(word)` , `x1(word)` , `y1(word)` , `x2(word)` , `y2(word)` , `colour(word)`

Commands	Description
cmd	0xFF7D
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.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x1(MSB), x1(LSB), y1(MSB), y1(LSB), x2(MSB), x2(LSB), y2(MSB),
// y2(LSB), colour(MSB), colour(LSB)

0xFF, 0x7D, 0x00, 0x0A, 0x00, 0x0F, 0x00, 0x28, 0x00, 0x50, 0x04, 0x10

// This will Line from X1=10 (Hex 0x00, 0x0A), Y1=15 (Hex 0x00, 0x0F), to X2=40
// (0x00, 0x28), Y2=80 (0x00, 0x50) of Colour=Teal (0x04, 0x10).

// The response will be 0x06 if the command is successful
```

5.2.6. Draw Rectangle

The **Draw Rectangle** command draws a rectangle 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 "Line Pattern" command.

Library Function: `gfx_Rectangle`

Syntax: `cmd(word)` , `x1(word)` , `y1(word)` , `x2(word)` , `y2(word)` , `colour(word)`

Commands	Description
cmd	0xFF7A
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.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x1(MSB), x1(LSB), y1(MSB), y1(LSB), x2(MSB), x2(LSB), y2(MSB),
// y2(LSB), colour(MSB), colour(LSB)

0xFF, 0x7A, 0x00, 0x0A, 0x00, 0x6E, 0x00, 0xC8, 0x00, 0xDC, 0xF8, 0x00

// The will draw a Rectangle from X1=10 (0x00, 0x0A), Y1=110 (0x00, 0x6E),
// to X2=200 (0x00, 0xC8), Y2=220 (0x00, 0xDC), of colour=Red (0xF8, 0x00).

// The response will be 0x06 if the command is successful
```

5.2.7. Draw Filled Rectangle

The Draw Filled Rectangle command draws a solid rectangle from x1, y1 to x2, y2 using the specified colour. The line may be tessellated with the "Line Pattern" command. The outline colour can be specified with the "Outline Colour" command. If "Outline Colour" is set to 0, no outline is drawn.

Library Function: `gfx_RectangleFilled`

Syntax: `cmd(word)` , `x1(word)` , `y1(word)` , `x2(word)` , `y2(word)` , `colour(word)`

Commands	Description
cmd	0xFF7A
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.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x1(MSB), x1(LSB), y1(MSB), y1(LSB), x2(MSB), x2(LSB), y2(MSB),
// y2(LSB), colour(MSB), colour(LSB)

0xFF, 0x79, 0x00, 0x32, 0x00, 0x3C, 0x00, 0x5A, 0x00, 0x64, 0x07, 0xE0

// The will draw a Solid Filled Rectangle from X1=50 (0x00, 0x32), Y1=60 (0x00, 0x3C),
// to X2=90 (0x00, 0x5A), Y2=100 (0x00, 0x64), of colour=Lime (0x07, 0xE0).

// The response will be 0x06 if the command is successful
```

5.2.8. Draw Polyline

The **Draw Polyline** command plots lines between points specified by a pair of arrays using the specified colour. The line may be tessellated with the "Line Pattern" command. The "Draw Polyline" command can be used to create complex raster graphics by loading the arrays from serial input or from MEDIA with very little code requirement.

Library Function: `gfx_Polyline`

Syntax: `cmd(word)` , `n(word)` , `vx1(word)` , `vxN(word)` , `vy1(word)` , `vyN(word)` , `colour(word)`

Commands	Description
cmd	0x0015
n	Specifies the number of elements in the x and y arrays specifying the vertices for the polyline.
vx, vy	Specifies the array of elements for the x/y coordinates of the vertices. Vx1, vx2, ..., vxN, vy1, vy2, ..., vyN
colour	Specifies the colour of the polyline.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), n(MSB), n(LSB), vx1(MSB), vx1(LSB), vx2(MSB), vx2(LSB), vx3(MSB),
// vx3(LSB), vy1(MSB), vy1(LSB), vy2(MSB), vy2(LSB), vy3(MSB), vy3(LSB), colour(MSB),
// colour(LSB)

0x00, 0x15, 0x00, 0x03, 0x00, 0x0A, 0x00, 0x50, 0x00, 0xB4, 0x00, 0x05, 0x00, 0xC8, 0x00,
0x50, 0x80, 0x00

// The following will draw a 3 point Polyline from X1=10 (0x00, 0x0A), Y1=5 (0x00, 0x05),
// to X2=80 (0x00, 0x50), Y2=200 (0x00, 0xC8), and finally to X3=180 (0x00, 0xB4),
// Y3=80 (0x00, 0x50) of Colour=Maroon (0x80, 0x00)

// The response will be 0x06 if the command is successful
```

5.2.9. Draw Polygon

The **Draw Polygon** command 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 line may be tessellated with the "Line Pattern" command. The Draw Polygon command can be used to create complex raster graphics by loading the arrays from serial input or from MEDIA with very little code requirement.

Library Function: `gfx_Polygon`

Syntax: `cmd(word)` , `n(word)` , `vx1(word)` , `vxN(word)` , `vy1(word)` , `vyN(word)` , `colour(word)`

Commands	Description
cmd	0x0013
n	Specifies the number of elements in the x and y arrays specifying the vertices for the polygon.
vx, vy	Specifies the array of elements for the x/y coordinates of the vertices. Vx1, vx2, ..., vxN, vy1, vy2, ..., vyN
colour	Specifies the colour of the polygon.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), n(MSB), n(LSB), vx1(MSB), vx1(LSB), vx2(MSB), vx2(LSB), vx3(MSB),
// vx3(LSB), vx4(MSB), vx4(LSB), vy1(MSB), vy1(LSB), vy2(MSB), vy2(LSB), vy3(MSB),
vy3(LSB),
// vy4(MSB), vy4(LSB), colour(MSB), colour(LSB)

0x00, 0x13, 0x00, 0x04, 0x00, 0x0A, 0x00, 0x50, 0x00, 0xB4, 0x00, 0xDC, 0x00, 0x05, 0x00,
0xC8, 0x00, 0x50, 0x00, 0x04, 0xFF, 0xE0

// The following will draw a 4 point Polyline from X1=10 (0x00, 0x0A), Y1=5 (0x00, 0x05),
// to X2=80 (0x00, 0x50), Y2=200 (0x00, 0xC8),  to X3=180 (0x00, 0xB4), Y3=80 (0x00, 0x50),
// and finally to X4=220 (0x00, 0xDC), Y4=4 (0x00, 0x04) of Colour=Yellow (0xFF, 0xE0)

// The response will be 0x06 if the command is successful
```

5.2.10. Draw Filled Polygon

The **Draw Filled Polygon** command 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 a minimum of 3 and can be specified in any fashion.

Library Function: `gfx_PolygonFilled`

Syntax: `cmd(word)`, `n(word)`, `vx1(word)`, `vxN(word)`, `vy1(word)`, `vyN(word)`, `colour(word)`

Commands	Description
cmd	0x0014
n	Specifies the number of elements in the x and y arrays specifying the vertices for the polygon.
vx, vy	Specifies the array of elements for the x/y coordinates of the vertices. Vx1, vx2, ..., vxN, vy1, vy2, ..., vyN
colour	Specifies the colour of the polygon.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), n(MSB), n(LSB), vx1(MSB), vx1(LSB), vx2(MSB), vx2(LSB), vx3(MSB),
// vx3(LSB), vx4(MSB), vx4(LSB), vy1(MSB), vy1(LSB), vy2(MSB), vy2(LSB), vy3(MSB),
// vy3(LSB), vy4(MSB), vy4(LSB), colour(MSB), colour(LSB)

0x00, 0x14, 0x00, 0x04, 0x00, 0x0A, 0x00, 0x50, 0x00, 0xB4, 0x00, 0xDC, 0x00, 0x05, 0x00,
0xC8, 0x00, 0x50, 0x00, 0x04, 0x04, 0x00

// The following will draw a 4 point Polyline from X1=10 (0x00, 0x0A), Y1=5 (0x00, 0x05),
// to X2=80 (0x00, 0x50), Y2=200 (0x00, 0xC8), to X3=180 (0x00, 0xB4), Y3=80 (0x00, 0x50),
// and finally to X4=220 (0x00, 0xDC), Y4=4 (0x00, 0x04) of Colour=Green (0x04, 0x00)

// The response will be 0x06 if the command is successful
```

5.2.11. Draw Triangle

The **Draw Triangle** command draws a triangle outline between vertices `x1,y1`, `x2,y2` and `x3,y3` using the specified colour. The line may be tessellated with the "Line Pattern" command.

Library Function: `gfx_Triangle`

Syntax: `cmd(word)`, `x1(word)`, `y1(word)`, `x2(word)`, `y2(word)`, `x3(word)`, `y3(word)`, `colour(word)`

Commands	Description
cmd	0xFF74
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 of the triangle.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x1(MSB), x1(LSB), y1(MSB), y1(LSB), x2(MSB), x2(LSB), y2(MSB),
// y2(LSB), x3(MSB), x3(LSB), y3(MSB), y3(LSB), colour(MSB), colour(LSB)

0xFF, 0x74, 0x00, 0x32, 0x00, 0x3C, 0x00, 0x14, 0x00, 0xAA, 0x00, 0x46, 0x00, 0xAA, 0x07,
0xFF

// This will draw a Triangle from X1=50 (0x00, 0x32), Y1=60 (0x00, 0x3C), to X2=20
// (0x00, 0x14), Y2=170 (0x00, 0xAA), to X3=70 (0x00, 0x46), Y3=170 (0x00, 0xAA) of
// colour=Aqua (0x07, 0xFF)

// The response will be 0x06 if the command is successful
```

5.2.12. Draw Filled Triangle

The **Draw Filled Triangle** command draws a solid triangle between vertices `x1, y1`, `x2, y2` and `x3, y3` using the specified colour.

Library Function: `gfx_TriangleFilled`

Syntax: `cmd(word)`, `x1(word)`, `y1(word)`, `x2(word)`, `y2(word)`, `x3(word)`, `y3(word)`, `colour(word)`

Commands	Description
cmd	0xFF59
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 of the triangle.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x1(MSB), x1(LSB), y1(MSB), y1(LSB), x2(MSB), x2(LSB), y2(MSB),
y2(LSB),
// x3(MSB), x3(LSB), y3(MSB), y3(LSB), colour(MSB), colour(LSB)

0xFF, 0x59, 0x00, 0x32, 0x00, 0x3C, 0x00, 0x14, 0x00, 0xAA, 0x00, 0x46, 0x00, 0xAA, 0x00,
0x1F

// This will draw a Triangle from X1=50 (0x00, 0x32), Y1=60 (0x00, 0x3C), to X2=20
// (0x00, 0x14), Y2=170 (0x00, 0xAA), to X3=70 (0x00, 0x46), Y3=170 (0x00, 0xAA) of
// colour=Blue (0x00, 0x1F)

// The response will be 0x06 if the command is successful
```

5.2.13. Calculate Orbit

The **Calculate Orbit** command 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 Xdest and Ydest.

Library Function: `gfx_Orbit`

Syntax: `cmd(word)`, `angle(word)`, `distance(word)`

Commands	Description
cmd	0x0012
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.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), angle(MSB), angle(LSB), distance(MSB), distance(LSB)

0x00, 0x12, 0x00, 0x28, 0x00, 0x3C

// This will calculate the x and y coordinates based on the Angle=40 degrees (0x00, 0x28)
// and the Distance=60 pixels (0x00, 0x3C) from the current origin.

// The response will be 0x06, 0x00, 0x2D, 0x00, 0x25 assuming the origin is at X=0, Y=0.
// New coordinates are X=45 (0x00, 0x2D) and Y=37 (0x00, 0x25)
```

5.2.14. Put pixel

The **Put Pixel** command draws a pixel at position x, y using the specified colour.

Library Function: `gfx_PutPixel`

Syntax: `cmd(word)` , `x(word)` , `y(word)` , `colour(word)`

Commands	Description
cmd	0xFF76
x, y	Specifies the pixel x, y coordinates.
colour	Specifies the colour of the pixel.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x(MSB), x(LSB), y(MSB), y(LSB), colour(MSB), colour(LSB)

0xFF, 0x76, 0x00, 0x28, 0x00, 0x64, 0xFF, 0xE0

// This will put a pixel at X=40 (0x00, 0x28), Y=100 (0x00, 0x64), and colour the pixel
// Yellow (0xFF, 0xE0).

// The response will be 0x06 if the command is successful.
```

5.2.15. Read Pixel

The **Read Pixel** command reads the colour value of the pixel at position x,y.

Library Function: `gfx_GetPixel`

Syntax: `cmd(word)` , `x(word)` , `y(word)`

Commands	Description
cmd	0xFF75
x, y	Specifies the pixel x, y coordinates.

Returns: `acknowledge(byte)` , `colour(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *colour*: 16bit colour of the pixel.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x(MSB), x(LSB), y(MSB), y(LSB)

0xFF, 0x75, 0x00, 0x28, 0x00, 0x64

// This will read the colour of a pixel at X=40 (0x00, 0x28), Y=100 (0x00, 0x64).

// The response will be 0x06, 0xFF, 0xE0 if the command is successful, assuming the pixel
// being read is coloured Yellow (0xFF, 0xE0).
```

5.2.16. Move Origin

The **Move Origin** command moves the origin to a new position, which is suitable for specifying the location for both graphics and text.

Library Function: `gfx_MoveTo`

Syntax: `cmd(word)`, `xpos(word)`, `ypos(word)`

Commands	Description
cmd	0xFF81
xpos, ypos	Specifies the horizontal and vertical position of the new origin.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), xpos(MSB), xpos(LSB), ypos(MSB), ypos(LSB)

0xFF, 0x81, 0x00, 0x32, 0x00, 0x5A

// This will move the Origin to be X=50 (0x00, 0x32), Y=90 (0x00, 0x5A)

// The response will be 0x06 if the command is successful.
```

5.2.17. Draw Line & Move Origin

The **Draw Line & Move Origin** command 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, using the "Set Graphics Parameters" – "Object Colour" command. The line may be tessellated with the "Line Pattern" command.

> **Note**
>
> This command is mostly useful with the "Calculate Orbit" command, and usually the "Draw Line" command would be used

Library Function: `gfx_LineTo`

Syntax: `cmd(word)`, `xpos(word)`, `ypos(word)`

Commands	Description
cmd	0xFF7F
xpos, ypos	Specifies the horizontal and vertical position of the line end as well as the new origin.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), xpos(MSB), xpos(LSB), ypos(MSB), ypos(LSB)

0xFF, 0x7F, 0x00, 0xC8, 0x00, 0xFA

// This will draw a line from the current origin (assuming this is X=0, Y=0 for this
// example) to X=200 (0x00, 0xC8), Y=250 (0x00, 0xFA) and set the origin to be this point
// (X=200, Y=250).

// The response will be 0x06 if the command is successful.
```

5.2.18. Clipping

The **Clipping** command Enables or Disables the ability for Clipping to be used. The clipping points are set with "Set Clip Window" and must be set first.

Library Function: `gfx_Clipping`

Syntax: `cmd(word)` , `value(word)`

Commands	Description
cmd	0xFF46
value	0 = Clipping Disabled, 1 = Clipping Enabled

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), value(MSB), value(LSB)

0xFF, 0x46, 0x00, 0x01

// This will Enable Clipping

// The response will be 0x06 if the command is successful.
```

5.2.19. Set Clip Window

The **Set Clip Window** command 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, the clipping setting must be enabled separately using the "Clipping" command

Library Function: `gfx_ClipWindow`

Syntax: `cmd(word)` , `x1(word)` , `y1(word)` , `x2(word)` , `y2(word)`

Commands	Description
cmd	0xFF6A
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.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x1(MSB), x1(LSB), y1(MSB), y1(LSB), x2(MSB), x2(LSB), y2(MSB),
y2(LSB)

0xFF, 0x6A, 0x00, 0x00, 0x00, 0x00, 0x00, 0x28, 0x00, 0x28

// This will set the top left of the Clipping Window Region to be X1=0 (0x00, 0x00), Y1=0
// (0x00, 0x00), and bottom right to be X2=40 (0x00, 0x28), Y2=40 (0x00, 0x28)

// The response will be 0x06 if the command is successful.
```

5.2.20. Extend Clip Region

The **Extend Clip Region** command forces the clip region to the extent of the last text that was printed, or the last image that was shown.

Library Function: `gfx_SetClipRegion`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFF68

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFF, 0x68

// This will extend the clip region to the extent of the last text or image that was shown.

// The response will be 0x06 if the command is successful.
```

5.2.21. Draw Ellipse

The **Draw Ellipse** command plots a coloured Ellipse on the screen at centre x, y with x-radius = xrad and y-radius = yrad.

Library Function: `gfx_Ellipse`

Syntax: `cmd(word)`, `x(word)`, `y(word)`, `xrad(word)`, `yrad(word)`, `colour(word)`,

Commands	Description
cmd	0xFF67
x,y	Specifies the horizontal and vertical position of the centre of ellipse.
xrad	Specifies x-radius of the ellipse.
yrad	Specifies y-radius of the ellipse.
colour	Specifies the colour of the ellipse.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x(MSB), x(LSB), y(MSB), y(LSB), xrad(MSB), xrad(LSB),
//  yrad(MSB), yrad(LSB), colour(MSB), colour(LSB)

0xFF, 0x67,0x00, 0x5A, 0x00, 0x3C, 0x00, 0x14, 0x00, 0x0F, 0xFF, 0xDE

// This will draw an Ellipse at X=90 (0x00, 0x5A), Y=60 (0x00, 0x3C), where the
// x-Radius is  20 (0x00, 0x14), and the y-Radius is 15 (0x00, 0x0F), where the colour
// is Cream (0xFF, 0xDE)

// The response will be 0x06 if the command is successful.
```

5.2.22. Draw Filled Ellipse

The **Draw Filled Ellipse** command plots a solid coloured Ellipse on the screen at centre x,y with x-radius = xrad and y-radius = yrad.

Library Function: `gfx_EllipseFilled`

Syntax: `cmd(word)`, `x(word)`, `y(word)`, `xrad(word)`, `yrad(word)`, `colour(word)`,

Commands	Description
cmd	0xFF66
x,y	Specifies the horizontal and vertical position of the centre of ellipse.
xrad	Specifies x-radius of the ellipse.
yrad	Specifies y-radius of the ellipse.
colour	Specifies the colour of the ellipse.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x(MSB), x(LSB), y(MSB), y(LSB), xrad(MSB), xrad(LSB), yrad(MSB),
// yrad(LSB), colour(MSB), colour(LSB)

0xFF, 0x66, 0x00, 0x5A, 0x00, 0x3C, 0x00, 0x14, 0x00, 0x0F, 0xFD, 0x20

// This will draw an Ellipse at X=90 (0x00, 0x5A), Y=60 (0x00, 0x3C), where the x-Radius
// is 20 (0x00, 0x14), and the y-Radius is 15 (0x00, 0x0F), where the colour is
// Orange (0xFD, 0x20)

// The response will be 0x06 if the command is successful.
```

5.2.23. Draw Button

The **Draw Button** command 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.

Library Function: `gfx_Button`

Syntax: `cmd(word)`, `state(word)`, `x(word)`, `y(word)`, `buttoncolour(word)`, `txtcolour(word)`, `font(word)`, `txtWidth(word)`, `txtHeight(word)`, `text (string)`

Commands	Description
cmd	0x0011
state	Appearance of button, 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. String must be Null terminated. char0, char1, char2, ..., charN, NULL

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), state(MSB), state(LSB), x(MSB), x(LSB), y(MSB), y(LSB),
// buttoncolour(MSB), buttoncolour(LSB), txtcolour(MSB), txtcolour(LSB), font(MSB),
font(LSB),
// txtWidth(MSB), txtWidth(LSB), txtHeight(MSB), txtHeight(LSB), char0, char1, char2,
char3,
// char4, char5, char6, char7, char8, NULL

x00, 0x11, 0x00, 0x00, 0x00, 0x50, 0x00, 0x50, 0x07, 0xFF, 0x90, 0x1A, 0x00, 0x01, 0x00,
0x01, 0x00, 0x01, 0x50, 0x72, 0x65, 0x73, 0x73, 0x20, 0x4D, 0x65, 0x00

// This will create a Button with the Up State being OFF, positioned at X=80 (0x00, 0x50),
// Y=80 (0x00, 0x50), where the Button Colour is Aqua (0x07, 0xFF), and the Text Colour is
// Dark Violet (0x90, 0x1A), the text Font is FONT_1 (0x00, 0x01), the Text Width multiplier
is
// 1 (0x00, 0x01), and the Text Height multiplier is also 1 (0x00, 0x01), and the Text
states
// “Press Me” and is Null Terminated.

// The response will be 0x06 if the command is successful.
```

5.2.24. Draw Panel

The **Draw Panel** command 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.

Library Function: `gfx_Panel`

Syntax: `cmd(word)` , `state(word)` , `x(word)` , `y(word)` , `Width(word)` , `Height(word)` , `colour(word)`

Commands	Description
cmd	0xFF5F
state	Appearance of panel, 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: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), state(MSB), state(LSB), x(MSB), x(LSB), y(MSB), y(LSB), Width(MSB),
// Width(LSB), Height(MSB), Height(LSB) colour(MSB), colour(LSB)

0xFF, 0x5F, 0x00, 0x01, 0x00, 0xC8, 0x00, 0xB4, 0x00, 0x01, 0x00, 0x01, 0xFF, 0x9C

// This will draw a Rectangular Panel which has a Raised Profile, located at X=200
// (0x00, 0xC8), Y=180 (0x00, 0xB4), where the Text Width multiplier is 1 (0x00, 0x01)
// and the Text Height multiplier is 1 (0x00, 0x01), and the colour is Linen (0xFF, 0x9C).

// The response will be 0x06 if the command is successful.
```

5.2.25. Draw Slider

The **Draw Slider** command draws a vertical or horizontal slider bar on the screen. The **Draw Slider** command 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:

- if x2-x1 > y2-y1 slider is assumed to be horizontal (ie: if width > height, slider is horizontal)
- if x2-x1 <= y2-y1 slider is assumed to be vertical (ie: if height <= width, slider is horizontal)
- 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)
- If value is negative, thumb is driven to the graphics position set by the ABSolute of value. (used to set thumb to its actual graphical position (usually by touch screen)
- The thumb colour is determine by the "Set Graphics Parameters" – "Object Colour" command, however, if the current object colour is BLACK, a darkened shade of the colour parameter is used for the thumb

Library Function: `gfx_Slider`

Syntax: `cmd(word)`, `mood(word)`, `x1(word)`, `y1(word)`, `x2(word)`, `y2(word)`, `colour(word)`, `scale(word)`, `value(word)`

Commands	Description
cmd	0xFF5E
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: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB), x1(MSB), x1(LSB), y1(MSB), y1(LSB), x2(MSB),
// x2(LSB), y2(MSB), y2(LSB), colour(MSB), colour(LSB), scale(MSB), scale(LSB), value(MSB),
// value(LSB)

0xFF, 0x5E, 0x00, 0x01, 0x00, 0x1E, 0x00, 0x28, 0x00, 0xD2, 0x00, 0x5A, 0x89, 0x5C, 0x00,
0x64, 0x00, 0x00

// This will create a Slider with a Raised Profile, with top left corner positioned at
// X1=30 (0x00, 0x1E), Y1=40 (0x00, 0x28), and bottom right corner positioned at X2=210
// (0x00, 0xD2), Y2=90 (0x00, 0x5A), where the slider colour is Blue Violet (0x89, 0x5C),
// Full scale is 100 (0x00, 0x64), and the value of the Thumb Slider is at 0 (0x00, 0x00).

// The response will be 0x06 if the command is successful.
```

5.2.26. Screen Copy Paste

The **Screen Copy Paste** command 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.

Library Function: `gfx_ScreenCopyPaste`

Syntax: `cmd(word)` , `xs(word)` , `ys(word)` , `xd(word)` , `yd(word)` , `width(word)` , `height(word)`

Commands	Description
cmd	0xFF5D
xs,ys	Specifies the horizontal 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.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), xs(MSB), xs(LSB), ys(MSB), ys(LSB), xd(MSB), xd(LSB), yd(MSB),
yd(LSB),
// width(MSB), width(LSB), height(MSB), height(LSB)

0xFF, 0x5D, 0x00, 0x0A, 0x00, 0x1E, 0x00, 0x5A, 0x01, 0x0E, 0x00, 0x5A, 0x00, 0x1E

// This will copy a section of the screen from X1=10 (0x00, 0x0A), Y1=30 (0x00, 0x1E) and
// paste it at X2=90 (0x00, 0x5A), Y2=270 (0x01, 0x0E), where the Width to copy/paste is
// 90 (0x00, 0x5A) and the Height is 30 (0x00, 0x1E)

// The response will be 0x06 if the command is successful.
```

5.2.27. Bevel Shadow

The **Bevel Shadow** command changes the graphics "Draw Button" commands bevel shadow depth.

Library Function: `gfx_BevelShadow`

Syntax: `cmd(word)` , `value(word)`

Commands	Description
cmd	0xFF3C
value	0 = No Bevel Shadow. 1-4 = Number of Pixels Deep (Default = 3).

Returns: `acknowledge(byte)` , `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: Previous Bevel Shadow status.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), value(MSB), value(LSB)

0xFF, 0x3C, 0x00, 0x02

// This will set the Bevel Shadow depth to be 2 pixels

// The response will be 0x06, 0x00, 0x03 assuming the previous Bevel Shadow Depth was
// set to 3 (0x00, 0x03) and if the command is successful.
```

5.2.28. Bevel Width

The **Bevel Width** command changes the graphics "Draw Button" commands bevel width.

Library Function: `gfx_BevelWidth`

Syntax: `cmd(word)`, `value(word)`

Commands	Description
cmd	0xFF3D
value	0 = No Bevel 1-15 = Number of Pixels Wide (Default = 2).

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: Previous Bevel Width status.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), value(MSB), value(LSB)

0xFF, 0x3D, 0x00, 0x0B

// This will set the Bevel Width to be 11 pixels

// The response will be 0x06, 0x00, 0x02 assuming the previous Bevel Shadow Depth
// was set to 2 (0x00, 0x04) and if the command is successful.
```

5.2.29. Background Colour

The **Background Colour** command sets the screen background colour.

Library Function: `gfx_BGcolour`

Syntax: `cmd(word)`, `colour(word)`

Commands	Description
cmd	0xFF48
colour	Specifies the colour to be set (0-65535 or HEX 0x0000-0xFFFF).

Returns: `acknowledge(byte)`, `colour(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *colour*: Previous Background Colour.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), colour(MSB), colour(LSB)

0xFF, 0x48, 0x00, 0x10

// This will set the Background Colour to be Navy (0x00, 0x10)

// The response will be 0x06, 0x00, 0x00 assuming the previous Background Colour was
// Black (0x00, 0x00) and if the command is successful.
```

5.2.30. Outline Colour

The **Outline Colour** command sets the outline colour for rectangles and circles.

Library Function: `gfx_OutlineColour`

Syntax: `cmd(word)`, `colour(word)`

Commands	Description
cmd	0xFF41
colour	Specifies the colour to be set (0-65535 or HEX 0x0000-0xFFFF), set to 0 for no effect.

Returns: `acknowledge(byte)`, `colour(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *colour*: Previous Outline Colour.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), colour(MSB), colour(LSB)

0xFF, 0x41, 0xF8, 0x1F

// This will set the Outline Colour to be Fuchsia (0xF8, 0x1F)

// The response will be 0x06, 0x00, 0x1F assuming the previous Outline Colour
// was Blue (0x00, 0x1F) and if the command is successful.
```

5.2.31. Contrast

The **Contrast** Command sets the contrast of the display, or turns it On/Off depending on display model.

Library Function: `gfx_Contrast`

Syntax: `cmd(word)`, `contrast(word)`

Commands	Description
cmd	0xFF40
contrast	All Diablo16 Display Modules supports Contrast values from 1-15 and 0 to turn the Display off.

Returns: `acknowledge(byte)`, `contrast(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Contrast value.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), contrast(MSB), contrast(LSB)

0xFF, 0x40, 0x00, 0x06

// This will set the Contrast of the display to be 6

// The response will be 0x06, 0x00, 0x00 assuming the previous Contrast
// was Display Off (0x00, 0x00) and if the command is successful.
```

5.2.32. Frame Delay

The **Frame Delay** command sets the inter frame delay for the "Media Video" command.

Library Function: `gfx_FrameDelay`

Syntax: `cmd(word)`, `Msec(word)`

Commands	Description
cmd	0xFF43
Msec	0-255 milliseconds.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Frame Delay value.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), Msec(MSB), Msec(LSB)

0xFF, 0x43, 0x00, 0x05

// This will set the Frame Delay to be 5 milliseconds

// The response will be 0x06, 0x00, 0x00 assuming the previous Frame Delay value
// was 0 (0x00, 0x00) and if the command is successful.
```

5.2.33. Line Pattern

The **Line Pattern** command sets the line draw pattern for line drawing. If set to zero, lines are solid, else each '1' bit represents a pixel that is turned off.

Library Function: `gfx_LinePattern`

Syntax: `cmd(word)`, `pattern(word)`

Commands	Description
cmd	0xFF3F
pattern	0-65535 (or HEX 0x0000-0xFFFF) = number of bits in the line are turned off to form a pattern.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Line Pattern value.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), pattern(MSB), pattern(LSB)

0xFF, 0x3F, 0x00, 0x08

// This will set the Line Pattern of the line to be drawn to have 8 bits out of the
// 65535 turned off.

// The response will be 0x06, 0x00, 0x00 assuming the previous Line Pattern value
// was 0 (0x00, 0x00) and if the command is successful.
```

5.2.34. Screen Mode

The **Screen Mode** command alters the graphics orientation LANDSCAPE, LANDSCAPE_R, PORTRAIT, PORTRAIT_R

Library Function: `gfx_ScreenMode`

Syntax: `cmd(word)` , `mode(word)`

Commands	Description
cmd	0xFF42
mode	0 = LANDSCAPE 1 = LANDSCAPE REVERSE 2 = PORTRAIT 3 = PORTRAIT REVERSE

Returns: `acknowledge(byte)` , `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Screen Mode value.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB)

0xFF, 0x42, 0x00, 0x00

// This will set the Screen Mode of the display to be Landscape.

// The response will be 0x06, 0x00, 0x02 assuming the previous Screen Mode value
// was Portrait (0x00, 0x02) and if the command is successful.
```

5.2.35. Transparency

The **Transparency** command turns the transparency ON or OFF. Transparency is automatically turned OFF after the next image or video command.

Library Function: `gfx_Transparency`

Syntax: `cmd(word)`, `mode(word)`

Commands	Description
cmd	0xFF44
mode	0 = Transparency OFF 1 = Transparency ON.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Transparency value.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB)

0xFF, 0x44, 0x00, 0x01

// This will set the Transparency of the display to be ON.

// The response will be 0x06, 0x00, 0x00 assuming the previous Transparency value
// was OFF (0x00, 0x00) and if the command is successful.
```

5.2.36. Transparent Colour

The **Transparent** Colour command alters the colour that needs to be made transparent.

Library Function: `gfx_TransparentColour`

Syntax: `cmd(word)`, `mode(word)`

Commands	Description
cmd	0xFF45
mode	0-65535 (or HEX 0x0000-0xFFFF) = colour to make transparent.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Previous Transparent Colour value.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB)

0xFF, 0x45, 0x84, 0x00

// This will set the Transparent Colour of the display to be Olive (0x84, 0x00).

// The response will be 0x06, 0x00, 0x00 assuming the previous Transparent Colour
// value was Black (0x00, 0x00) and if the command is successful.
```

5.2.37. Set Graphics Parameters

Returns various graphics parameters to the caller.

Library Function: `gfx_Set`

Syntax: `cmd(word)`, `function(word)`, `mode(word)`

Commands	Description
cmd	0xFF83
function	18 Object Colour Sets the Object colour used in various functions such as Draw Slider and Draw Line & Move Origin.
value	0 – 65535 or 0 - 0xFFFF

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), function(MSB), function(LSB), value(MSB), value(LSB)

0xFF, 0x83, 0x00, 0x12, 0x04, 0x00

// This will call the Object Colour (18 = 0x00, 0x012) command and set the object
// colour to be Green (0x04, 0x00)

// The response will be 0x06 if successful
```

5.2.38. Get Graphics Parameters

Returns various graphics parameters to the caller.

Library Function: `gfx_Get`

Syntax: `cmd(word)`, `mode(word)`

Commands	Description
cmd	0xFF4A
mode	mode = 0: Current orientations maximum X value (X_MAX) mode = 1: Current orientations maximum Y value (Y_MAX) mode = 2: Left location of last Object mode = 3: Top location of Object mode = 4: Right location of last Object mode = 5: Bottom location of Object

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: **Mode0**: Returns the maximum horizontal resolution of the display, minus 1. X_MAX returns Horizontal Resolution - 1

  Mode1: Returns the maximum vertical resolution of the display, minus 1. Y_MAX returns Vertical Resolution - 1

  Mode2: Returns the left location of the last drawn object

  Mode3: Returns the top location of the last drawn object

  Mode4: Returns the right location of the last drawn object

  Mode5: Returns the bottom location of the last drawn object

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB)

0xFF, 0x4A, 0x00, 0x01

// This will request the display current maximum Y value based on the screens orientation.

// The response will be 0x06, 0x00, 0xEF which is ACK followed by 239 (0x00, 0xEF)
// assuming the display is in Landscape mode, with 239 Pixels in the Y Direction.
// The return is 0 based, so it's the resolution – 1.
```

5.3. Media Commands (SD/SDHC Memory Cards)

The following is a summary of the commands available to be used for Media:

- Media Init
- Set Byte Address
- Set Sector Address
- Read Sector
- Write Sector
- Read Byte
- Read Word
- Write Byte
- Write Word
- Flush Media
- Display Image (RAW)
- Display Video (RAW)
- Display Video Frame (RAW)

5.3.1. Media Init

The **Media Init** command initialises a uSD/SD/SDHC memory card for further operations. The SD card is connected to the SPI (serial peripheral interface) of the Diablo16 Processor.

Library Function: `media_Init`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFF25

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: 1 if memory card is present and successfully initialised.
  0 if no card is present or not able to initialise.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFF, 0x25

// This command will initialize a uSD/SD/SDHC memory card so it can be used for
// further operations.

// The response will be 0x06 if the command is successful.
```

5.3.2. Set Byte Address

The **Sey Byte Address** command sets the media memory internal Address pointer for access at a non-sector aligned byte address.

Library Function: `media_SetAdd`

Syntax: `cmd(word)`, `HIword(word)`, `LOword(word)`

Commands	Description
cmd	0xFF2F
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.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), HIword(MSB), HIword(LSB), LOword(MSB), LOword(LSB)

0xFF, 0x2F, 0x00, 0x00, 0x02, 0x01

// This will set the media address to byte 513 (0x00, 0x00, 0x02, 0x01)
// (which is sector #1, 2nd byte in sector) for subsequent operations.

// The response will be 0x06 if the command is successful.
```

5.3.3. Set Sector Address

The Set **Sector Address** command sets the media memory internal Address pointer for sector access.

Library Function: `media_SetSector`

Syntax: `cmd(word)`, `HIword(word)`, `LOword(word)`

Commands	Description
cmd	0xFF2E
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.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), HIword(MSB), HIword(LSB), LOword(MSB), LOword(LSB)

0xFF, 0x2E, 0x00, 0x00, 0x00, 0x0A

// This will set the media address to the 11th (0x00, 0x00, 0x00, 0x0A)
// sector (which is also byte address 5120) for subsequent operations.

// The response will be 0x06 if the command is successful.
```

5.3.4. Read Sector

The **Read Sector** command reads and returns 512 bytes (256 words) pointed to by the internal Sector pointer, determined by the "Set Sector Address" command. After the read the Sector pointer is automatically incremented by 1.

Library Function: `media_RdSector`

Syntax: `cmd(word)`

Commands	Description
cmd	0x0016

Returns: `acknowledge(byte)`, `status(word)`, `block(sector)`

Example

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1** for successful media response, else **0** for attempt failed.
- *block*: 512 bytes (256 words)

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0x00, 0x16

// This will initiate the read and return of 512 bytes starting where the
// Set Sector Address command was set to.

// The response will be 0x06 if the command is successful.
```

See also

See also the "Media Init" command to enable the media to be ready for access, and "Set Sector Address" command to define where reading is to occur.

5.3.5. Write Sector

The **Write Sector** command writes 512 bytes (256 words) from a source memory block into the uSD card. After the write the Sect pointer is automatically incremented by 1.

Library Function: `media_WrSector`

Syntax: `cmd(word)`, `block(sector)`

Commands	Description
cmd	0x0017
block	512 bytes (256 words) to be written to the media sector address.

Returns: `acknowledge(byte)`, `status(word)`

Example

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1** for successful media response, else **0** for attempt failed.

```
// Byte Stream:
// cmd(MSB), cmd(LSB), block(sector)

0x00, 0x17, 0x...(512 Bytes worth of data)...

// This will transfer a 512 bytes block of data to the address pointed to by the
// Set Sector Address command.

// The response will be 0x06 if the command is successful.
```

See also

See also the "Media Init" command to enable the media to be ready for access, and "Set Sector Address" command to define where reading is to occur.

5.3.6. Read Byte

The **Read Byte** command returns the byte value from the current media address, set by the "Set Byte Address" command. The internal byte address will then be internally incremented by one.

Library Function: `media_ReadByte`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFF2B

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Byte value in the LSB.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFF, 0x2B

// This will read and return the byte value from the media address set by the
// Set Byte Address command.

// The response will be 0x06, 0x00, 0xFF assuming the value being read was
// 255 (0x00, 0xFF). Due to the Diablo16 being a 16bit system, each byte is reported
// in word format (2 bytes).
```

See also

See also the "Media Init" command to enable the media to be ready for access, and "Set Byte Address" command to define where reading is to occur.

5.3.7. Read Word

The **Read Word** command returns the word value (2 bytes) from the current media address, set by the "Set Byte Address" command. The internal byte address will then be internally incremented by one. If the address is not aligned, the word will still be read correctly.

Library Function: `media_ReadWord`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFF2A

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Word value.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFF, 0x2A

// This will read and return the byte value from the media address set by the
// Set Byte Address command.

// The response will be 0x06, 0x3B, 0xAF assuming the value being read was
// 15279 (0x3B, 0xAF).
```

See also

See also the "Media Init" command to enable the media to be ready for access, and "Set Byte Address" command to define where reading is to occur.

5.3.8. Write Byte

Writes a byte to the current media address that was initially set with the "Set Sector Address" command.

Note

Writing bytes or words to a media sector must start from the beginning of the sector. All writes will be incremental until the "Flush Media" command is executed, or the sector address rolls over to the next sector. When the "Flush Media" command is called, any remaining bytes in the sector will be padded with 0xFF, destroying the previous contents. An attempt to use the "Set Byte Address" command will result in the lower 9 bits being interpreted as zero. If the writing rolls over to the next sector, the "Flush Media" command is issued automatically internally.

Library Function: `media_WriteByte`

Syntax: `cmd(word)`, `value(word)`

Commands	Description
cmd	0xFF29
value	Byte value, in the LSB, to be written at the current byte address location.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **Non-Zero** for successful media response, else **0** for attempt failed

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), value(MSB), value(LSB)

0xFF, 0x29, 0x00, 0x61

// This will write the ASCII character 'a' (0x00, 0x61) as a byte to the media address
// set by Set Sector Address.

// The response will be 0x06, 0x00, 0x01 assuming the value being written was successful.
```

See also

See also the "Media Init" command to enable the media to be ready for access, and "Set Sector Address" command to define where reading is to occur.

5.3.9. Write Word

Writes a word to the current media address that was initially set with the "Set Sector Address" command.

Note

Writing bytes or words to a media sector must start from the beginning of the sector. All writes will be incremental until the "Flush Media" command is executed, or the sector address rolls over to the next sector. When "Flush Media" command is called, any remaining bytes in the sector will be padded with 0xFF, destroying the previous contents. An attempt to use the "Set Byte Address" command will result in the lower 9 bits being interpreted as zero. If the writing rolls over to the next sector, the "Flush Media" command is issued automatically internally

Library Function: `media_WriteWord`

Syntax: `cmd(word)`, `value(word)`

Commands	Description
cmd	0xFF28
value	The 16 bit word to be written at the current media address location.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **Non-Zero** for successful media response, else **0** for attempt failed

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), value(MSB), value(LSB)

0xFF, 0x28, 0x00, 0x41

// This will write the ASCII character 'A' (0x00, 0x41) as a word to the media address set
// by Set Sector Address.

// The response will be 0x06, 0x00, 0x01 assuming the value being written was successful.
```

See also

See also the "Media Init" command to enable the media to be ready for access, and "Set Sector Address" command to define where reading is to occur.

5.3.10. Flush Media

After writing any data to a sector, the **Flush Media** command 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.

Library Function: `media_Flush`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFF26

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **Non-Zero** for successful media response, else 0 for attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFF, 0x26

// This command will ensure data written to the current sector is correctly stored to
// the media.

// The response will be 0x06, 0xFF, 0xFF if the command is successful (see Status above)
```

5.3.11. Display Image (RAW)

Displays an image from the media storage at the specified co-ordinates. The image address is previously specified with the "Set Byte Address" command or "Set Sector Address" command. If the image is shown partially off-screen, it may not be displayed correctly.

Library Function: `media_Image`

Syntax: `cmd(word)` , `x(word)` , `y(word)`

Commands	Description
cmd	0xFF27
x,y	Specifies the top left position where the image will be displayed.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x(MSB), x(LSB), y(MSB), y(LSB)

0xFF, 0x27, 0x00, 0x0A, 0x00, 0x14

// This will display an image at X=10 (0x00, 0x0A), Y=20 (0x00, 0x14) from the media
storage
// location specified.

// The response will be 0x06 if the command is successful.
```

See also

See also the "Media Init" command to enable the media to be ready for access, and "Set Byte Address" or "Set Sector Address" command to define where reading is to occur.

5.3.12. Display Video (RAW)

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 "Set Byte Address" or "Set Sector Address" commands. 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 "Display Video Frame" command for alternatives

Library Function: `media_Video`

Syntax: `cmd(word)`, `x(word)`, `y(word)`

Commands	Description
cmd	0xFF31
x,y	Specifies the top left position where the video clip will be displayed.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x(MSB), x(LSB), y(MSB), y(LSB)

0xFF, 0x31, 0x00, 0x32, 0x00, 0x0A

// This will display a video clip at X=50 (0x00, 0x32), Y=10 (0x00, 0x0A) from the media
storage
// device location specified.

// The response will be 0x06 if the command is successful.
```

See also

See also the "Media Init" command to enable the media to be ready for access, and "Set Byte Address" or "Set Sector Address" command to define where reading is to occur. See the "Display Video Frame" command for an alternative.

5.3.13. Display Video Frame (RAW)

Displays a video from the media storage device at the specified co-ordinates. The video address is previously specified with the "Set Byte Address" command or "Set Sector Address" command. 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

The **Display Video Frame (RAW)** command will now show an 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.

Library Function: `media_VideoFrame`

Syntax: `cmd(word)`, `x(word)`, `y(word)`, `frameNumber(word)`

Commands	Description
cmd	0xFF30
x,y	Specifies the top left position of the video frame to be displayed.
frameNumber	Specifies the required frame number to be displayed.

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response

Example:

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x(MSB), x(LSB), y(MSB), y(LSB), frameNumber(MSB), frameNumber(LSB)

0xFF, 0x30, 0x00, 0x23, 0x00, 0x05, 0x00, 0x2D

// This will display frame number 45 (0x00, 0x2D) of the video clip stored at the address
// specified, and display it at location X=35 (0x00, 0x23), Y=5 (0x00, 0x05).

// The response will be 0x06 if the command is successful.
```

See also

See also the "Media Init" command to enable the media to be ready for access, and "Set Byte Address" or "Set Sector Address" command to define where reading is to occur.

5.4. Serial (UART) Communications Commands

The following is a summary of the commands available to be used for Serial (UART) Communications:

- Set Baud Rate

5.4.1. Set Baud Rate

The **Set Baud Rate** command is used to set the required baud rate. To set the default baud rate, please refer to the instructions in Introduction to Workshop4 - Serial Environment.

Library Function: `setbaudWait`

Syntax: `cmd(word)`, `index(word)`

Commands	Description
cmd	0x0026
index	Specifies the baud rate index value. Refer to the baud rate index table below.

index	Required Baud Rate	%Error	Actual Baud Rate
0	110	0.00%	110
1	300	0.00%	300
2	600	0.00%	600
3	1200	0.00%	1200
4	2400	0.04%	2401
5	4800	0.04%	4802
6	9600	0.16%	9615
7	14400	0.27%	14439
8	19200	0.38%	19273
9	31250	0.00%	31250
10	38400	0.83%	38717
11	56000	0.16%	56090
12	57600	1.27%	58333
13	115200	2.64%	118243
14	128000	0.53%	128676
15	256000	0.53%	257353
16	300000	4.17%	312500
17	375000	6.06%	397727
18	500000	9.38%	546875
19	600000	4.17%	625000

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example:

```
// Byte Stream:
// cmd(MSB), cmd(LSB), index(MSB), index(LSB)

0x00, 0x26, 0x00, 0x0D

// This will set the baud rate to be 115200, which is Index 13 (0x00, 0x0D)
// The response will be 0x06 at the new baud rate set, 100ms after the command is sent.
```

5.5. Timer Commands

The following is a summary of the commands available to be used for the Timers:

- Sleep

5.5.1. Sleep

The **Sleep** command puts 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 touched. When returning from sleep mode, the display and processor are restored from low power mode.

> **Note**
>
> Prior to PmmC R33, the Sleep command units were not approximately a second in length. This was fixed in R33.

Library Function: `sys_Sleep`

Syntax: `cmd(word)`, `units(word)`

Commands	Description
cmd	0xFE6D
units	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. Sleep timer units may vary, however 1 unit is approximately 1 second.

Returns: `acknowledge(byte)`, `units(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *units*: Remaining time units when touch screen is touched, else returns zero.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), units(MSB), units(LSB)

0xFE, 0x6D, 0x00, 0x0A

// This will put the display to sleep for 10 (0x00, 0x0A) ‘units’, or approximately 10
// seconds. If the display is touched in this time, it will return the number of ‘units’
// remaining in the timer.

// The response is 0x06, 0x00, 0x00 assuming the display was not touched during this
period.
```

5.6. FAT16 File Commands

The following is a summary of the commands available to be used for FAT16:

- File Error
- File Count
- List Filenames
- Find First File
- Find First File and Report
- Find Next File
- Find Next File and Report
- Find Exists
- File Open
- File Close
- File Read
- File Seek
- File Index
- File Tell
- File Write
- File Size
- Display Image (FAT)
- Screen Capture
- Write Character to the File
- Read Character from the File
- Write Word to the File
- Read Word from the File
- Write String to the File
- Read String from the File
- File Erase
- File Rewind
- File Load Function

- File Call Function
- File Run
- File Execute
- Load Image Control
- File Mount
- File Unmount
- Play WAV File
- Load String for 4XE/4FN File
- Read String for 4XE/4FN File

5.6.1. File Error

Returns the most recent error code or 0 if there were no errors.

Library Function: `file_Error`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFE58

Returns: `acknowledge(byte)`, `ErrorNumber(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *ErrorNumber*: Error Number. Refer to the Error table below for more details.

ErrorNumber	Description
1	IDE command execution error
2	CARD not present
3	WRONG partition type, not FAT16
4	MBR sector invalid signature
5	Boot Record invalid signature
6	Media not mounted
7	File not found in open for read
8	File not open
9	Fat attempt to read beyond EOF
10	Reached the end of file
11	Invalid cluster value > maxcls
12	All root dir entry are taken
13	All clusters in partition are taken
14	A file with same name exist already
15	Cannot init the CARD
16	Cannot read the MBR
17	Malloc could not allocate the FILE struct
18	Mode was not r.w.
19	Failure during FILE search
20	Invalid Filename
21	bad media
22	Sector Read fail
23	Sector write fail

Example:

```
// Byte Stream:
// cmd(MSB), cmd(LSB), line(MSB), line(LSB), column(MSB), column(LSB)

0xFE, 0x58

// This will request the most recent error code from the display.

// The response will be 0x06, 0x00, 0x02 assuming the most recent error was 2 (0x00, 0x02)
// Card not Present.
```

5.6.2. File Count

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.

Library Function: `file_Count`

Syntax: `cmd(word)`, `filename(string)`

Commands	Description
cmd	0x0001
filename	Name of the file(s) for the search (passed as a string).Filename must be 8.3 format. char0, char1, char2, ..., charN, NULL.

Returns: `acknowledge(byte)`, `count(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *count*: Number of files that match the criteria.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), char0, char1, char2, NULL

0x00, 0x01, 0x2A, 0x2E, 0x2A, 0x00

// This will request the display to return the number of files on the disk, by sending the
// string “*.*” (0x2A, 0x2E, 0x2A) followed by a NULL.

// The response will be 0x06, 0x00, 0x23 assuming there are 35 (0x00, 0x23) files located
on
// the root of the micro SD card.
```

See also

The “File Mount” command, to initially mount the file system.

5.6.3. List Filenames

Lists the stream of file names that agree with the search key on the Display Screen. 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.

Note

"Find First File and Report" and "Find Next File and Report" are recommended alternatives in order to return the responses.

Library Function: `file_Dir`

Syntax: `cmd(word)`, `filename(string)`

Commands	Description
cmd	0x0002
filename	Name of the file(s) for the search (passed as a string).Filename must be 8.3 format. char0, char1, char2, ..., charN, NULL.

Returns: `acknowledge(byte)`, `count(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *count*: Number of files that match the criteria.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), char0, char1, char2, char3, char4, NULL

0x00, 0x02, 0x2A, 0x2E, 0x34, 0x58, 0x45, 0x00

// This will list on the display all the files on the root of the uSD card that fall in the
// category of "*.4XE" (0x2A, 0x2E, 0x34, 0x58, 0x45) followed by a NULL.

// The response will be 0x06, 0x00, 0x03 assuming there are 3 (0x00, 0x03) files located on
// the root of the micro SD card with the extension *.4XE

// The listing of these 3 files will also be displayed on the screen.
```

See also

The "File Mount" command, to initially mount the file system.
"Find First File and Report" and "Find Next File and Report" commands as alternatives which return the responses.

5.6.4. Find First File

Returns true if at least 1 file exists that satisfies the file argument. Wildcards are usually used so if the "Find First File" command returns true, further tests can be made using the "Find Next File" command to find all the files that match the wildcard class. Note that the filename is printed on the screen.

> **Note**
>
> "Find First File and Report" and "Find Next File and Report" are recommended alternatives in order to return the responses

Library Function: `file_FindFirst`

Syntax: `cmd(word)`, `filename(string)`

Commands	Description
cmd	0x0006
filename	Name of the file(s) for the search (passed as a string).Filename must be 8.3 format. char0, char1, char2, ..., charN, NULL.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If at least one file exists that satisfies the criteria, else **0**: If no file satisfies the criteria.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), char0, char1, char2, char3, char4, NULL

0x00, 0x06, 0x2E, 0x2A, 0x47, 0x43, 0x49, 0x00

// This will list on the display the first file on the root of the uSD card that falls in
// the category of "*.GCI" (0x2E, 0x2A, 0x47, 0x43, 0x49) followed by a NULL.

// The response will be 0x06, 0x00, 0x01 assuming there was at least 1 (0x00, 0x01)
// file located on the root of the micro SD card that satisfied this search.The listing of
this
// file will also be displayed on the screen.
```

> **See also**
>
> The "File Mount" command, to initially mount the file system.
> "Find Next File" command, to find the next file which meets the criteria.
> "Find First File and Report" and "Find Next File and Report" commands as alternatives which return the responses.

5.6.5. Find First File and Report

The **Find First File and Report** command returns the length of the filename and the filename if at least 1 file exists that matches the criteria.

Wildcards are usually used so if **Find First File and Report** command returns the *stringlength* and *filename*, further tests can be made using "Find Next File" or "Find Next File and Report" commands to find all the files that match the wildcard class.

Library Function: `file_FindFirstRet`

Syntax: `cmd(word)` , `filename(string)`

Commands	Description
cmd	0x0024
filename	Name of the file(s) for the search (passed as a string). Filename must be 8.3 format. char0, char1, char2, ..., charN, NULL.

Returns: `acknowledge(byte)` , `stringlength(word)` , `filename(string)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *stringlength*: Length of the File-name string.
- *filename*: Filename if it exists. Filename string is not NULL terminated.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), char0, char1, char2, char3, char4, NULL

0x00, 0x24, 0x2E, 0x2A, 0x47, 0x43, 0x49, 0x00

// This will list on the display the first file on the root of the uSD card that falls in
the
// category of "*.GCI" (0x2A, 0x2E, 0x47, 0x43, 0x49) followed by a NULL.

// The response will be 0x06, 0x00, 0x07, 0x42, 0x6F, 0x62, 0x2A, 0x47, 0x43, 0x49 assuming
// there was a file in the root of the uSD card called "Bob.GCI", where the reported length
of
// the filename was 7 (0x00, 0x07), and the filename was reported "Bob.GCI" (0x42, 0x6F,
0x62,
// 0x2E, 0x47, 0x43, 0x49).
```

See also

The "File Mount" command, to initially mount the file system.
"Find Next File and Report" and "Find Next File" commands to find the next file which meets the criteria.

5.6.6. Find Next File

The **Find Next File** command returns true if more file exists that satisfies the file argument that was given for the "Find First File" or "Find First File and Report" commands. Wildcards must be used for the "Find First File" or "Find First File and Report" commands else this function will always return zero as the only occurrence will have already been found.

Note that the filename is printed on the screen.

Library Function: `file_FindNext`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFE54

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If at least one file exists that satisfies the criteria, else **0**: If no file satisfies the criteria.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFE, 0x54

// This will find the next file that meets the criteria specified in the Find First File or
// Find First File and Report commands used previously.

// The response will be 0x06, 0x00, 0x01 assuming there is another file found that matches
// the criteria.
```

See also

The "File Mount" command, to initially mount the file system.
"Find First File" command, to find the next file which meets the criteria.
"Find First File and Report" and "Find Next File and Report" commands as alternatives which return the responses.

5.6.7. Find Next File and Report

Returns length of the filename and the filename if at least 1 file exists that matches the criteria given for the "Find First File" or "Find First File and Report" commands. Wildcards must be used for the "Find First File" or "Find First File and Report" commands else this function will always return zero as the only occurrence will have already been found.

Wildcards are usually used, so if the "Find First File" or "Find First File and Report" commands return the stringlength and filename, further tests can be made using **Find Next File and Report** command to find all the files that match the wildcard class.

Library Function: `file_FindNextRet`

Syntax: `cmd(word)`

Commands	Description
cmd	0x0025

Returns: `acknowledge(byte)`, `stringlength(word)`, `filename(string)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *stringlength*: Length of the File-name string.
- *filename*: Filename if it exists. Filename string is not NULL terminated.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0x00, 0x25

// This will find the next file that meets the criteria specified in the Find First File or
// Find First File and Report commands used previously.

// The response will be 0x06, 0x00, 0x07, 0x42, 0x6F, 0x62, 0x2E, 0x47, 0x43, 0x49 assuming
// there was a file in the root of the uSD card that matched the wild card search criteria
// used in the “Find First File” or “Find First File and Report” commands, where the
reported
// length of the filename was 7 (0x00, 0x07), and the filename was reported “Bob.GCI”
// (0x42, 0x6F, 0x62, 0x2E, 0x47, 0x43, 0x49).
```

See also

The "File Mount" command, to initially mount the file system.
"Find First File and Report" and "Find First File" commands to find the next file which meets the criteria.

5.6.8. File Exists

Tests for the existence of the file provided with the search key. Returns TRUE if found.

Library Function: `file_Exists`

Syntax: `cmd(word)`, `filename(string)`

Commands	Description
cmd	0x0005
filename	Name of the file(s) for the search (passed as a string). Filename must be 8.3 format. char0, char1, char2, ..., charN, NULL.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: File found, else **0**: File not found

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), char0, char1, char2, char3, char4, char5, char6, char7, NULL

0x00, 0x05, 0x54, 0x45, 0x53, 0x54, 0x2E, 0x34, 0x58, 0x45, 0x00

// This will search for the file “TEST.4XE” (0x54, 0x45, 0x53, 0x54, 0x2E, 0x34, 0x58,
0x45)
// on the uSD card, the string is ended with a NULL (0x00).

// The response will be 0x06, 0x00, 0x01 assuming the file was found.
```

See also

The “File Mount” command, to initially mount the file system.

5.6.9. File Open

Returns handle if file exists. The file 'handle' that is created is now used as reference for 'filename' for further file commands such as "File Close", 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 (i.e. no errors) and can be read with the "File Error" command.
For File Read mode ('r') the file must exist else a null handle (0x00, 0x00) is returned and the 'file not found' error number is set which can be read with the "File Error" command.

> **Note**
>
> If a file is opened for File 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.

Library Function: `file_Open`

Syntax: `cmd(word)`, `filename(string)`, `mode(byte)`

Commands	Description
cmd	0x000A
filename	Name of the file(s) for the search (passed as a string). Filename must be 8.3 format. char0, char1, char2, ..., charN, NULL.
mode	r' or 0x72 for File Read 'w' or 0x77 for File Write 'a' or 0x61 for File Append

Returns: `acknowledge(byte)`, `handle(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *handle*: Returns handle if file exists. Sets internal file error number accordingly (0 if no errors).

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), line(MSB), line(LSB), column(MSB), column(LSB)

0x00, 0x0A, 0x54, 0x45, 0x53, 0x54, 0x2E, 0x54, 0x58, 0x54, 0x00, 0x72

// This will attempt to read (0x72) a file called "TEST.TXT" (0x54, 0x45, 0x53, 0x54, 0x2E,
// 0x54, 0x58, 0x54) followed by a NULL (0x00) from the uSD Card

// The response will be 0x06, 0x14, 0x65 assuming the command was a success and the handle
// that was created had the value of DEC 5221 (0x14, 0x65).
```

See also

The “File Mount” command, to initially mount the file system.
The “File Close” command, to close the file once opened with this command.

5.6.10. File Close

The **File Close** command will close the previously opened file.

Library Function: `file_Close`

Syntax: `cmd(word)`, `handle(word)`

Commands	Description
cmd	0xFE51
handle	The file handle that was created by the "File Open" command which is now used as reference 'handle' for the filename, for further file functions such as in this function to close the file.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: File Closed, else **0**: File not closed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB)

0xFE, 0x51, 0x14, 0x65

// This will close the file with the handle value of 5221 (0x14, 0x65) which was
// opened previously

// The response will be 0x06, 0x00, 0x01 assuming the command was a success and
// the file was successfully closed.
```

See also

The "File Mount" command, to initially mount the file system.
The "File Open" command, to initially open the file.

5.6.11. File Read

Returns the number of bytes specified by **'size'** from the file referenced by **'handle'**.

Library Function: `file_Read`

Syntax: `cmd(word)`, `size(word)`, `handle(word)`

Commands	Description
cmd	0x000C
size	Number of bytes to be read.
handle	The handle that references the file to be read.

Returns: `acknowledge(byte)`, `count(word)`, `data(string)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *count*: Returns the number of bytes read.
- *data*: Data read from the file

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB)

0x00, 0x0C, 0x00, 0x14, 0x14, 0x65

// This will read 20 bytes (0x00, 0x14) from the file with handle 5221 (0x14, 0x65)

// The response will be 0x06, 0x00, 0x14, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
// 0x38, 0x39, 0x30, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A assuming
// the command was a success, and 20 bytes (0x00, 0x14) were read. The File contained
// the following data: 1234567890abcdefghij
```

See also

The "File Mount" command, to initially mount the file system.

5.6.12. File Seek

The **File Seek** command 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, e.g. assuming the file size is 10000 bytes, the **File Seek** command with HiWord = 0x00 and LoWord = 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 "Read Character from the File", "Read Word from the File", "Read String from the File" commands, or an image can be displayed with the "Display Image (FAT)" command. Conversely, "Write Character to the File", "Write Word to the File", "Write String to the File" commands 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, visible from the "File Error" command.

Library Function: `file_Seek`

Syntax: `cmd(word)`, `handle(word)`, `HiWord(word)`, `LoWord (word)`

Commands	Description
cmd	0xFE4F
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: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If Seek successful, else **0**: if attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), HiWord(MSB), HiWord(LSB), LoWord(MSB),
// LoWord(LSB)

0xFE, 0x4F, 0x10, 0xD5, 0x00, 0x00, 0x12, 0x34

// This will place a file pointer at the byte position 4660 (HiWord = 0x00, 0x00,
// LoWord = 0x12, 0x34) on the file with handle 4309 (0x10, 0xD5)

// The response will be 0x06, 0x00, 0x01 if the command was successful and the
// Seek was successful.
```

See also

The "File Mount" command, to initially mount the file system.
The "Read Character from the File", "Read Word from the File", "Read String from the File", "Write Character to the File", "Write Word to the File", and "Write String to the File" commands.
"Display Image (FAT)" command for displaying the image from File.
"File Error" command for retrieving any error which may have occurred.

5.6.13. File Index

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, e.g. assuming the record size is 100 bytes, the **File Index** command with HiSize = 0, LoSize = 100 and recordnum = 22 will set the file position to 2200 for the file handle, so subsequent data may be read from that position onwards with "Read Character from the File", "Read Word from the File", "Read String from the File" commands or an image can be displayed with the "Display Image (FAT)" command.
Conversely, the "Write Character to the File", "Write Word to the File", "Write String to the File" commands 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, visible from the "File Error" command.

Library Function: `file_Index`

Syntax: `cmd(word)`, `handle(word)`, `HiSize(word)`, `LoSize(word)`, `recordnum(word)`

Commands	Description
cmd	0xFE4E
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: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If the index found successfully, else **0**: if the attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), HiSize(MSB), HiSize(LSB), LoSize(MSB),
// LoSize(LSB), recordnum(MSB), recordnum(LSB)

0xFE, 0x4E, 0x10, 0xD5, 0x00, 0x00, 0x00, 0x64, 0x00, 0x16

// This will place a file pointer at the end of the file records specified, 22 records
// where each record is of size 100, (HiSize = 0x00, 0x00, LoSize = 0x00, 0x64,
// recordnum = 0x00, 0x16) on the file with handle 4309 (0x10, 0xD5)

// The response will be 0x06, 0x00, 0x01 if the command was successful and the Index
// was successful.
```

See also

The "File Mount" command, to initially mount the file system.
The "Read Character from the File", "Read Word from the File", "Read String from the File", "Write Character to the File", "Write Word to the File", and "Write String to the File" commands.
"Display Image (FAT)" command for displaying the image from File.
"File Error" command for retrieving any error which may have occurred.

5.6.14. File Tell

The **File Tell** command returns the current value of the file pointer.

Library Function: `file_Tell`

Syntax: `cmd(word)`, `handle(word)`

Commands	Description
cmd	0x000F
handle	The handle that references the file.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If the operation successful, else **0**: if the attempt failed.
- *HiWord*: Contains the upper 16bits of the value of the pointer
- *LoWord*: Contains the lower 16bits of the value of the pointer

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB)

0x00, 0x0F, 0x10, 0xD5

// This will return the current value of the file pointer 4309 (0x10, 0xD5)

// The response will be 0x06, 0x00, 0x01, 0x00, 0x00, 0x08, 0x98 assuming the command
// was successful (0x06), the operation was successful (0x00, 0x01), and the file pointer
// had the value of 2200 (0x00, 0x00, 0x08, 0x98)
```

See also

The "File Mount" command, to initially mount the file system.

5.6.15. File Write

The **File Write** command returns the current value of the file pointer.

Library Function: `file_Write`

Syntax: `cmd(word)`, `size(word)`, `source(string)`, `handle(word)`

Commands	Description
cmd	0x0010
size	Number of bytes to be written. Maximum that can be written at one time is 512 bytes.
source	String of Data without Null terminator.
handle	The handle that references the file to write.

Returns: `acknowledge(byte)`, `count(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *count*: 1: **Returns the number of bytes written**.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), size(MSB), size(LSB), source(MSB), source(LSB), handle(MSB),
// handle(LSB)

0x00, 0x10, 0x00, 0x05, 0x48, 0x65, 0x6C, 0x6C, 0x6F, 0x0F, 0xB8

// This will write 5 bytes (0x00, 0x05) where the string of data is “Hello” (0x48, 0x65,
// 0x6C, 0x6C, 0x6F) to the file with the handle of 4024 (0x0F, 0xB8)

// The response will be 0x06, 0x00, 0x05 assuming the command was successful and 5 bytes
// (0x00, 0x05) were successfully written
```

See also

The “File Mount” command, to initially mount the file system.

5.6.16. File Size

The **File Size** command reads the 32 bit file size.

Library Function: `file_Size`

Syntax: `cmd(word)`, `handle(word)`

Commands	Description
cmd	0x000E
handle	The handle that references the file to write.

Returns: `acknowledge(byte)`, `status(word)`, `HiWord(word)`, `LoWord(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If the operation successful, else **0**: if the attempt failed.
- *HiWord*: Contains the upper 16bits of the file size.
- *LoWord*: Contains the lower 16bits of the file size.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB)

0x00, 0x0E, 0x0F, 0xB8

// This will request the size of the file with the handle 4024 (0x0F, 0xB8)

// The response will be 0x06, 0x00, 0x01, 0x00, 0x00, 0x00, 0xA7 assuming the command was
// uccessful (0x06), the operation was successful (0x00, 0x01), and the file size was 167
// (0x00, 0x00, 0x00, 0xA7)
```

See also

The "File Mount" command, to initially mount the file system.

5.6.17. Display Image (FAT)

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 the "File Seek" command

Library Function: `file_Image`

Syntax: `cmd(word)` , `x(word)` , `y(word)` , `handle(word)`

Commands	Description
cmd	0xFE4A
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: `acknowledge(byte)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *error*: Returns a copy of the File Error, see the "File Error" command

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x(MSB), x(LSB), y(MSB), y(LSB), handle(MSB), handle(LSB)

0xFE, 0x4A, 0x00, 0x05, 0x00, 0x05, 0x0E, 0x9B

// This will display the image which has the file handle of 3739 (0x0E, 0x9B) at position
// X=5 (0x00, 0x05), Y=5 (0x00, 0x05)

// The response will be 0x06, 0x00, 0x00 if the command was successful and there was no
// error associated with this command.
```

See also

The "File Mount" command, to initially mount the file system.
"File Seek" command to access another image from the same file, if required.
"File Error" command for retrieving any error which may have occurred.

5.6.18. Screen Capture

The **Screen Capture** command saves an image of the screen shot to file at the current file position. The image can later be displayed with the "Display Image (FAT)" command. The file may be opened in append mode to accumulate multiple images. Later, the images can be displayed with the "File Seek" command. The image is saved from x, y (with respect to top left corner), and the capture area is determined by "width" and "height".

Library Function: `file_ScreenCapture`

Syntax: `cmd(word)`, `x(word)`, `y(word)`, `width(word)`, `height(word)`, `handle(word)`

Commands	Description
cmd	0xFE49
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: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: 0: If the operation was successful

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x(MSB), x(LSB), y(MSB), y(LSB), width(MSB), width(LSB), height(MSB),
// height(LSB), handle(MSB), handle(LSB)

0xFE, 0x49, 0x00, 0x00, 0x00, 0x00, 0x00, 0x64, 0x00, 0x64, 0x0C, 0x4E

// This will capture from X=0 (0x00, 0x00), Y=0 (0x00, 0x00) across 100 pixels (0x00, 0x64)
// and down 100 pixels (0x00, 0x64), and save the image inside that region to the file with
// handle 3150 (0x0C, 0x4E)

// The response will be 0x06, 0x00, 0x00 if the command was successful (0x06) and the
// operation was successful (0x00, 0x00)
```

See also

The "File Mount" command, to initially mount the file system.
"Display Image (FAT)" command for displaying the image from File.
"File Seek" command to access another image from the same file, if required.

5.6.19. Write Character to the File

This function writes the byte specified by "char" to the file, at the position indicated by the associated file-position pointer (set by the "File Seek" or "File Index" commands) and advances the pointer appropriately (incremented by 1). The file must be previously opened with 'w' (write) or 'a' (append) modes.

Library Function: `file_PutC`

Syntax: `cmd(word)`, `char(word)`, `handle(word)`

Commands	Description
cmd	0x001F
char	Data byte (in the LSB) about to be written.
handle	The handle that references the file to be written to.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: Returns the number of bytes written successfully

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), char(MSB), char(LSB), handle(MSB), handle(LSB)

0x00, 0x1F, 0x00, 0x58, 0x0B, 0x31

// This will write the character 'X' (0x00, 0x58) to the file with handle 2865 (0x0B, 0x31)

// The response will be 0x06, 0x00, 0x01 if the command was successful (0x06) and the
// operation successfully wrote the 1 byte (0x00, 0x01)
```

See also

The "File Mount" command, to initially mount the file system.
"File Seek" and "File Index" commands to access another image from the same file, if required.

5.6.20. Read Character from the File

The **Read Character from the File** command reads a byte from the file, at the position indicated by the associated file-position pointer (set by the "File Seek" or "File Index" commands) and advances the pointer appropriately (incremented by 1). The file must be previously opened with 'r' (read) mode.

Library Function: `file_GetC`

Syntax: `cmd(word)`, `handle(word)`

Commands	Description
cmd	0xFE47
handle	The handle that references the file to be read from.

Returns: `acknowledge(byte)`, `char(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *char*: Returns the data byte read from the file in the LSB.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB)

0xFE, 0x47, 0x0B, 0x31

// This will read the character from the file with the point of 2865 (0x0B, 0x31) based on
// the position of the pointer determined previously by the "File Seek" or "File Index"
// commands.

// The response will be 0x06, 0x00, 0x74 assuming the command was successful and the
pointer
// was pointing at the position of the file which contained the character 't' (0x00, 0x74)
```

See also

The "File Mount" command, to initially mount the file system.
"File Seek" and "File Index" commands to access another image from the same file, if required.

5.6.21. Write Word to the File

This function writes word sized (2 bytes) data specified by 'word' to the file, at the position indicated by the associated file-position pointer (set by the "File Seek" or "File Index" commands) and advances the pointer appropriately (incremented by 2). The file must be previously opened with 'w' (write) or 'a' (append) modes.

Library Function: `file_PutW`

Syntax: `cmd(word)`, `word(word)`, `handle(word)`

Commands	Description
cmd	0xFE46
word	Word about to be written.
handle	The handle that references the file to be written to.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: Returns the number of bytes written successfully

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), word(MSB), word(LSB), handle(MSB), handle(LSB)

0xFE, 0x46, 0x01, 0xBB, 0x0B, 0x31

// This will write the word 443 (0x01, 0xBB) to the file with handle 2865 (0x0B, 0x31).

// The response will be 0x06, 0x00, 0x02 assuming the command was successful and the
// operation was successful at writing the 2 bytes (0x00, 0x02).
```

See also

The "File Mount" command, to initially mount the file system.
"File Seek" and "File Index" commands to access another image from the same file, if required.

5.6.22. Read Word from the File

This function reads a word (2 bytes) from the file, at the position indicated by the associated file-position pointer (set by the "File Seek" or "File Index" commands) and advances the pointer appropriately (incremented by 2). The file must be previously opened with 'r' (read) mode

Library Function: `file_GetW`

Syntax: `cmd(word)`, `handle(word)`

Commands	Description
cmd	0xFE45
handle	The handle that references the file to be read from.

Returns: `acknowledge(byte)`, `word(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *word*: Returns the word read from the file.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB)

0xFE, 0x45, 0x0B, 0x31

// This will read the character from the file with the point of 2865 (0x0B, 0x31) based on
// the position of the pointer determined previously by the "[File Seek](#file-seek)" or
// "[File Index](#file-index)" commands.

// The response will be 0x06, 0x00, 0x74 assuming the command was successful and the
pointer
// was pointing at the position of the file which contained the word 25972 (0x65, 0x74)
```

See also

The "File Mount" command, to initially mount the file system.
"File Seek" and "File Index" commands to access another image from the same file, if required.

5.6.23. Write String to the File

This function writes a null terminated string to the file, at the position indicated by the associated file-position pointer (set by the "File Seek" or "File Index" commands) and advances the pointer appropriately. The file must be previously opened with 'w' (write) or 'a' (append) modes.

Library Function: `file_PutS`

Syntax: `cmd(word)`, `data(string)`, `handle(word)`

Commands	Description
cmd	0x0020
data	A Null terminated string to be written to the file.
handle	The handle that references the file to be written to.

Returns: `acknowledge(byte)`, `count(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *count*: Returns the number of characters written (excluding the null terminator).

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), char0, char1, char2, char3, char4, char5, char6, char7, char8,
// handle(MSB), handle(LSB)

0x00, 0x20, 0x34, 0x44, 0x20, 0x53, 0x79, 0x73, 0x74, 0x65, 0x6D, 0x73, 0x00, 0x0B, 0x31

// This will write the string "4D Systems" (0x34, 0x44, 0x20, 0x53, 0x79, 0x73, 0x74, 0x65,
// 0x6D, 0x73) followed by a Null (0x00) to the file which has a handle of 2865 (0x0B, 0x31)

// The response will be 0x06, 0x00, 0x0A assuming the command was successful and the 10
// characters (0x00, 0x0A) were written
```

See also

The "File Mount" command, to initially mount the file system.
"File Seek" and "File Index" commands to access another image from the same file, if required.

5.6.24. Read String from the File

This function reads a line of text from a file at the current file position indicated by the associated file-position pointer (set by the "File Seek" or "File Index" commands) 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.

Library Function: `file_GetS`

Syntax: `cmd(word)`, `size(word)`, `handle(word)`

Commands	Description
cmd	0x0007
size	The maximum number of bytes to be read from the file.
handle	The handle that references the file to be read from.

Returns: `acknowledge(byte)`, `word(word)`, `data(string)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *count*: Returns the number of characters read from file (excluding the null teminator).
- *data*: Returns the string read from the file excluding the Null terminator.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), size(MSB), size(LSB), handle(MSB), handle(LSB)

0x00, 0x07, 0x00, 0x05, 0x0B, 0x31

// This will read the string from the file with handle 2865 (0x0B, 0x31) up to the maximum
// of 5 characters (0x00, 0x05) in length.

// The response will be 0x06, 0x00, 0x04, 0x74, 0x65, 0x73, 0x74 assuming the command was
// successful and the file contained only 4 characters (0x00, 0x04) at the pointer
location,
// and the string was "test" (0x74, 0x65, 0x73, 0x74)
```

See also

The "File Mount" command, to initially mount the file system.
"File Seek" and "File Index" commands to access another image from the same file, if required.

5.6.25. File Erase

This function erases a file on the disk.

> **Note**
>
> If the function fails, the appropriate error number is set in the "File Error" command and will usually be error 19, "failure during FILE search".

Library Function: `file_Erase`

Syntax: `cmd(word)`, `filename(string)`

Commands	Description
cmd	0x0003
filename	Name of the file to be erased (passed as a string). Filename must be 8.3 format. char0, char1, char2, ..., charN, NULL

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If the operation successful, else **0**: if the attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), char0, char1, char2, char3, char4, char5, char6, char7, NULL

0x00, 0x03, 0x74, 0x65, 0x73, 0x74, 0x2E, 0x74, 0x78, 0x74, 0x00

// This will erase the file called "test.txt" (0x74, 0x65, 0x73, 0x74, 0x2E, 0x74, 0x78,
// 0x74) followed by NULL (0x00)

// The response will be 0x06, 0x00, 0x01 assuming the command was successful and the
// operation was successful
```

> **See also**
>
> The "File Mount" command, to initially mount the file system.
> "File Error" command for retrieving any error which may have occurred.

5.6.26. File Rewind

The **File Rewind** command resets the file pointer to the beginning of a file that has been opened in **'r'** (read), **'w'**, or **'a'** (append) mode.

Library Function: `file_Rewind`

Syntax: `cmd(word)`, `handle(word)`

Commands	Description
cmd	0xFE41
handle	The handle that references the file.

Returns: `acknowledge(byte)`, `word(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If the operation successful, else **0**: if the attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB)

0xFE, 0x41, 0x0B, 0x31

// This will reset the file point to the beginning of the file with file pointer 2865
// (0x0B, 0x31)

// The response will be 0x06, 0x00, 0x01 assuming the command was successful and the
// operation was successful
```

See also

The "File Mount" command, to initially mount the file system.

5.6.27. File Load Function

The **File Load Function** command allocates the RAM area to the 4FN or 4XE program, load it from the uSD card in to the RAM 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 using the "File Call Function" commands. The 4FN or 4XE program may be discarded at any time when no longer required, thus freeing its memory resources.

The loaded function can be discarded with the "Memory Free" command.

> **Note**
>
> A 4FN or a 4XE file is an executable file generated when a 4DGL file is compiled. 4DGL file refers to the program files developed under "Designer" or "ViSi" Environments in the 4D Workshop4 IDE.

4FN file is generated when the 4DGL program has 'main' with arguments.

4XE file is generated when the 4DGL program has a 'main', with no arguments.

Library Function: `file_LoadFunction`

Syntax: `cmd(word)`, `filename(string)`

Commands	Description
cmd	0x0008
filename	Name of the 4DGL function (filename.4FN) or application program (filename.4XE) that is about to be loaded into RAM. Filename must be 8.3 format. char0, char1, char2, ..., charN, NULL

Returns: `acknowledge(byte)`, `pointer(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *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.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), char0, char1, char2, char3, char4, char5, char6, char7, char8,
char9,
// char10, char11, NULL

0x00 0x08 0x34 0x46 0x4E 0x2D 0x50 0x52 0x4F 0x47 0x2E 0x34 0x46 0x4E 0x00

// This will load the "4FN-Prog.4FN" (0x34 0x46 0x4E 0x2D 0x50 0x52 0x4F 0x47 0x2E 0x34
0x46
// 0x4E 0x00) file, followed by a NULL.

// The response will be 0x06, 0x0D, 0x8B assuming the command was successful and the
```

```
pointer
// in memory where the function call has been loaded is 3467 (0x0D, 0x8B)
```

See also

The “File Mount” command, to initially mount the file system.
“File Call Function” command to invoke a loaded function
“Memory Free” command to discard a loaded function

5.6.28. File Call Function

Call the function previously loaded through "File Load Function".

Parameters may be passed to it in a conventional way except the strings which needs to be loaded in to memory location separately through "Load String for 4XE/4FN File" command and the string handle is given to the File Call Function. The 4FN function or 4XE application may be discarded at any time when no longer required, thus freeing its memory resources.

The loaded function can be discarded with the "Memory Free" command.

> **Note**
>
> A 4FN or a 4XE file is an executable file generated when a 4DGL file is compiled.

4FN file is generated when the 4DGL program has 'main' with arguments.
4XE file is generated when the 4DGL program has a 'main', with no arguments.

Library Function: `file_CallFunction`

Syntax: `cmd(word)`, `handle(word)`, `Argcount(word)`, `Arg0(word)`, `Arg1(word)`, `...`, `ArgN(word)`

Commands	Description
cmd	0x0008
handle	The file handle that was created by the "File Load Function" command which is now used as reference 'handle' for the filename, for further file functions such as in this function to close the file.
Argcount	Number of arguments to be passed to the File Run command. Maximum 6 arguments
Arg0	Argument 0 to be passed. (optional)
Arg1	Argument 1 to be passed. (optional)
ArgN	Argument N to be passed. (optional)

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Returns the value from main in the called function

Examples

4DGL Program

```
// This program “4FN-Prog.4FN” when compiled under the “Designer Environment”
// generates the .4FN file.

#platform "uLCD-70DT"
#inherit "4DGL_16bitColours.fnc"

/* A 4DGL program without 'main'. When compiled, a .4FN extension file is generated at the
root folder where the 4DGL program resides. Copy the 4FN file to the Fat16 (aka FAT)
formatted uSD card.*/

func messagebox(var line, var col, var txt)
    var txts ;
    gfx_Cls();
    gfx_ScreenMode(PORTRAIT) ;          // Change Orientation
    print("I am the Child Program\n") ; // Print text on screen
    print("line=", line, "\n");      // Print the 1st parameter
    print("column=", col, "\n");     // Print the 2nd parameter

    txt_MoveCursor(line, col);      // Move cursor to line, col
    txts := txt ;              // because str_Printf changes txt
    str_Printf(&txt, "%s");          // Print the 3rd parameter
    pause(3000);                        // Pause for 3 sec.
    str_Copy(txts,"I have returned");
    return;
endfunc
```

File Mount command

```
// cmd(MSB), cmd(LSB)
0xFF, 0x03
// Response:
0x06 0x15 0x43 //( ACK, Status(MSB), Status(LSB) )
```

File Load command

```
// cmd(MSB), cmd(LSB), char0, char1, char2, char3, char4, char5, char6, char7, char8,
char9,
// char10, char11, NULL
0x00 0x08 0x34 0x46 0x4E 0x2D 0x50 0x52 0x4F 0x47 0x2E 0x34 0x46 0x4E 0x00
// Response:
0x06 0x95 0x52 //( ACK, Pointer(MSB), Pointer(LSB) )
```

Load String command

```
// Cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), char0, char1, char2, char3, char4, char5,
char6,
```

```
// char7, char8, char9, char10, NULL
0x00 0x21 0x00 0x00 0x48 0x65 0x6C 0x6C 0x6F 0x20 0x57 0x6F 0x72 0x6C 0x64 0x00
// Response:
0x06 0x01 0x0E //( ACK, pointer(MSB), pointer(LSB) )
```

File Call command (Arg0 = 10, Arg1 = 10, Arg2 = String Pointer)

```
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), Argcount(MSB), Argcount(LSB), Arg0(MSB),
Arg0(LSB),
// Arg1(MSB), Arg1(LSB), Arg2(MSB), Arg2(LSB)
0x00 0x19 0x95 0x52 0x00 0x03 0x00 0x0A 0x00 0x0A 0x01 0x0E
// Response:
0x06 0x00 0x00 //( ACK, value(MSB), value(LSB) )
```

Read String command

```
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB)
0x00 0x22 0x01 0x0E
// Response:
0x49 0x20 0x68 0x61 0x76 0x65 0x20 0x72 0x65 0x74 0x75 0x72 0x6E 0x65 0x64
// ( ACK, char0, char1, char2, char3, char4, char5, char6, char7, char8, char9, char10,
char11,
// char12, char13, char14, char15, char16 )

// Response = “I have returned”
```

See also

The “File Mount” command, to initially mount the file system.
“File Load Function" command to load a function
“Memory Free” command to discard a loaded function
“Load String for 4XE/4FN File” command to pass a string to the Function

5.6.29. File Run

The **File Run** command will load the 4FN or 4XE program from the uSD card in to the RAM and execute it. Once the program is called, the Host must wait until the program finished execution. Any attempt to send further commands while the 4FN or 4XE file is executing can cause the module to reset or respond with erroneous data.

The 4FN or 4XE program may be discarded at any time when no longer required, thus freeing its memory resources.

Parameters may be passed to it in a conventional way except the strings which needs to be loaded in to memory location separately through "**Load String**" command and the string handle is given to the File Call Function. The 4FN function or 4XE application may be discarded at any time when no longer required, thus freeing its memory resources.
The loaded function can be discarded with the "Memory Free" command.

> **Note**
>
> A 4FN or a 4XE file is an executable file generated when a 4DGL file is compiled.
> 4FN file is generated when the 4DGL program has 'main' with arguments.
> 4XE file is generated when the 4DGL program has a 'main', with no arguments.

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 Argcount is 0, no arguments are passed; else Arg0-ArgN contains argument 0 to argument N.

The disk does not need to be mounted; File Run automatically mounts the drive.

Library Function: `file_Run`

Syntax: `cmd(word)`, `filename(string)`, `Argcount(word)`, `Arg0(word)`, `Arg1(word)`, `...`, `ArgN(word)`

Commands	Description
cmd	0x000D
filename	A 4FN or a 4XE file is an executable file generated when a 4DGL file is compiled. Filename must be 8.3 format. char0, char1, char2, ..., charN, NULL
Argcount	Number of arguments to be passed to the File Run command
Arg0	Argument 0 to be passed. (optional)
Arg1	Argument 1 to be passed. (optional)
ArgN	Argument N to be passed. (optional)

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Returns the value from the called program.

Examples

4DGL Program

```
// This program “4FN-Prog.4FN” when compiled under the “Designer Environment”
// generates the .4FN file.

#platform "uLCD-70DT"
#inherit "4DGL_16bitColours.fnc"

/* A 4DGL program without 'main'. When compiled, a .4FN extension file is
generated at the root folder where the 4DGL program resides. Copy the 4FN file
to the Fat16 (aka FAT) formatted uSD card.*/

func messagebox(var line, var col, var txt)
    var txts ;
    gfx_Cls();
    gfx_ScreenMode(PORTRAIT) ;          // Change Orientation
    print("I am the Child Program\n") ; // Print text on screen
    print("line=", line, "\n");      // Print the 1st parameter
    print("column=", col, "\n");     // Print the 2nd parameter

    txt_MoveCursor(line, col);      // Move cursor to line, col
    txts := txt ;              // because str_Printf changes txt
    str_Printf(&txt, "%s");          // Print the 3rd parameter
    pause(3000);                        // Pause for 3 sec.
    str_Copy(txts,"I have returned");
    return;
endfunc
```

File Mount command

```
cmd(MSB), cmd(LSB)
0xFF, 0x03
// Response:
0x06 0x15 0x43 ( ACK, Status(MSB), Status(LSB) )
```

Load String command

```
// Cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), char0, char1, char2, char3, char4,
// char5, char6, char7, char8, char9, char10, NULL
0x00 0x21 0x00 0x00 0x48 0x65 0x6C 0x6C 0x6F 0x20 0x57 0x6F 0x72 0x6C 0x64 0x00
// Response:
0x06 0x01 0x0E //( ACK, pointer(MSB), pointer(LSB) )
```

File Run command (Arg0 = 10, Arg1 = 10, Arg2 = String Pointer)

```
// cmd(MSB), cmd(LSB), char0, char1, char2, char3, char4, char5, char6, char7, char8,
// char9, char10, char11, Argcount(MSB), Argcount(LSB), Arg0(MSB), Arg0(LSB), Arg1(MSB),
// Arg1(LSB), Arg2(MSB), Arg2(LSB)
```

```
0x00 0x0D 0x34 0x46 0x4E 0x2D 0x50 0x52 0x4F 0x47 0x2E 0x34 0x46 0x4E 0x00 0x00 0x03 0x00
0x0A 0x00 0x0A 0x01 0x0E
// Response:
0x06 0x80 0x24
```

Read String command

```
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB)
0x00 0x22 0x01 0x0E
// Response:
0x49 0x20 0x68 0x61 0x76 0x65 0x20 0x72 0x65 0x74 0x75 0x72 0x6E 0x65 0x64
// ( ACK, char0, char1, char2, char3, char4, char5, char6, char7, char8, char9, char10,
// char11, char12, char13, char14, char15, char16 )

// Response = “I have returned”
```

5.6.30. File Execute

The **File Execute** command will load the 4FN or 4XE program from the uSD card in to the RAM and execute it. Once the program is called, the Host must wait until the program finished execution. Any attempt to send further commands while the 4FN or 4XE file is executing can cause the module to reset or respond with erroneous data.

The 4FN or 4XE program may be discarded at any time when no longer required, thus freeing its memory resources.

Parameters may be passed to it in a conventional way except the strings which needs to be loaded in to memory location separately through "**Load String**" command and the string handle is given to the File Call Function. The 4FN function or 4XE application may be discarded at any time when no longer required, thus freeing its memory resources.

The loaded function can be discarded with the "Memory Free" command.

> **Note**
>
> A 4FN or a 4XE file is an executable file generated when a 4DGL file is compiled.
> 4FN file is generated when the 4DGL program has 'main' with arguments.
> 4XE file is generated when the 4DGL program has a 'main', with no arguments.

This function is similar to File Run, however, the main program in FLASH retains all memory allocations (e.g. file buffers, memory allocated with mem_Alloc etc.)

Library Function: `file_Exec`

Syntax: `cmd(word)`, `filename(string)`, `Argcount(word)`, `Arg0(word)`, `Arg1(word)`, `...`, `ArgN(word)`

Commands	Description
cmd	0x0004
filename	A 4FN or a 4XE file A 4FN or a 4XE file is an executable file generated when a 4DGL file is compiled. Filename must be 8.3 format. char0, char1, char2, ..., charN, NULL
Argcount	Number of arguments to be passed to the File Run command
Arg0	Argument 0 to be passed. (optional)
Arg1	Argument 1 to be passed. (optional)
ArgN	Argument N to be passed. (optional)

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Returns the value from the called program.

Examples

4DGL Program

```
// This program “4FN-Prog.4FN” when compiled under the “Designer Environment”
// generates the .4FN file.

#platform "uLCD-70DT"
#inherit "4DGL_16bitColours.fnc"

/* A 4DGL program without 'main'. When compiled, a .4FN extension file is
generated at the root folder where the 4DGL program resides. Copy the 4FN file
to the Fat16 (aka FAT) formatted uSD card.*/

func messagebox(var line, var col, var txt)
    var txts ;
    gfx_Cls();
    gfx_ScreenMode(PORTRAIT) ;          // Change Orientation
    print("I am the Child Program\n") ; // Print text on screen
    print("line=", line, "\n");      // Print the 1st parameter
    print("column=", col, "\n");     // Print the 2nd parameter

    txt_MoveCursor(line, col);      // Move cursor to line, col
    txts := txt ;              // because str_Printf changes txt
    str_Printf(&txt, "%s");          // Print the 3rd parameter
    pause(3000);                        // Pause for 3 sec.
    str_Copy(txts,"I have returned");
    return;
endfunc
```

File Mount command

```
cmd(MSB), cmd(LSB)
0xFF, 0x03
// Response:
0x06 0x15 0x43 ( ACK, Status(MSB), Status(LSB) )
```

Load String command

```
// Cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), char0, char1, char2, char3, char4, char5,
// char6, char7, char8, char9, char10, NULL
0x00 0x21 0x00 0x00 0x48 0x65 0x6C 0x6C 0x6F 0x20 0x57 0x6F 0x72 0x6C 0x64 0x00
// Response:
0x06 0x01 0x0E //( ACK, pointer(MSB), pointer(LSB) )
```

File Execute command (Arg0 = 10, Arg1 = 10, Arg2 = String Pointer)

```
// cmd(MSB), cmd(LSB), char0, char1, char2, char3, char4, char5, char6, char7, char8,
char9,
// char10, char11, Argcount(MSB), Argcount(LSB), Arg0(MSB), Arg0(LSB), Arg1(MSB),
```

```
Arg1(LSB),
// Arg2(MSB), Arg2(LSB)
0x00 0x04 0x34 0x46 0x4E 0x2D 0x50 0x52 0x4F 0x47 0x2E 0x34 0x46 0x4E 0x00 0x00 0x03 0x00
0x0A 0x00 0x0A 0x01 0x0E
// Response:
0x06 0x80 0x24
```

Read String command

```
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB)
0x00 0x22 0x01 0x0E
// Response:
0x49 0x20 0x68 0x61 0x76 0x65 0x20 0x72 0x65 0x74 0x75 0x72 0x6E 0x65 0x64
// ( ACK, char0, char1, char2, char3, char4, char5, char6, char7, char8, char9, char10,
char11,
// char12, char13, char14, char15, char16 )

// Response = “I have returned”
```

5.6.31. Load Image Control

Reads a control file to create an image list. The GCI file may contain images, videos or animations built through the Graphics Composer Software tool.

The GCI file is created by selecting the **GCI – FAT Selected Folder** option in the Built Option type. See the Graphics Composer User Guide for further details on the Graphics Composer.

Library Function: `file_LoadImageControl`

Syntax: `cmd(word)`, `filename1(string)`, `filename2(string)`, `mode(word)`

Commands	Description
cmd	0x0009
filename1	The control list filename "*.dat". Created from Graphics Composer. Filename must be 8.3 format. char0, char1, char2, ..., charN, NULL
filename2	The image filename "*.gci". Created from Graphics Composer. Filename must be 8.3 format. char0, char1, char2, ..., charN, NULL
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 **Load Image Control** command can open the file (*.gci) and use the "File Seek" command to get to the position of the image which can then automatically be displayed using the "Display Image (FAT)" command. 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 the "File Seek" command 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 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: `acknowledge(byte)`, `handle(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *handle*: Returns a handle (pointer to the memory allocation) to the image control list that has been created else NULL if function fails.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), charA0, charA1, charA2, ..., charA12, NULL, charB0, charB1, charB2,
// ..., char12, NULL, mode(MSB), mode(LSB)

0x00, 0x09, 0x47, 0x46, 0x58, 0x32, 0x44, 0x45, 0x4D, 0x4F, 0x2E, 0x44, 0x41, 0x54, 0x00,
0x47, 0x46, 0x58, 0x32, 0x44, 0x45, 0x4D, 0x4F, 0x2E, 0x47, 0x43, 0x49, 0x00, 0x00, 0x01

// This will load the Image Control System using the 2 specified files (GFX2DEMO.DAT and
// GFX2DEMO.GCI)

// The response will be 0x06 0x0D 0x6A assuming the command is successful and the handle
// that is returned is 3434 (0x0D, 0x6A)
```

See also

The "File Mount" command, to initially mount the file system.
"File Seek" command to access another image from the same file, if required.
"Display Image (FAT)" command for displaying the image from File.

5.6.32. File Mount

Starts up the FAT16 disk file services and allocates a small 32 byte control block for subsequent use. When you open a file using the "File Open" command a further 512 + 44 = 556 bytes are attached to the FAT16 file control block. When you close a file using the "File Close" command, the 556 byte allocation is released leaving the 32 byte file control block. The **File Mount** command 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 the "File Unmount" command.

Library Function: `file_Mount`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFE3C

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Non-zero: If the operation successful.
  0: if the attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFE, 0x3C

// This will mounts the file system

// The response will be 0x06 followed by a non-zero number (such as 0x00, 0x01) if the
// command is successful, or zero (0x00, 0x00) if unsuccessful.
```

See also

The "File Unmount" command, to unmount the file system.

5.6.33. File Unmount

The "File Unmount" command releases any buffers for FAT16 and unmount the Disk File System. This function is to be called to close the FAT16 file system.

Library Function: `file_Unmount`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFE3B

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFE, 0x3B

// This will unmounts the file system

// The response will be 0x06 if the command is successful
```

See also

The "File Mount" command, to initially mount the file system.

5.6.34. Play WAV File

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 Commands” for additional play control functions.

> **Note**
>
> Wave files should be mono to keep data bandwidth to a minimum, and should be 'canonic' format. Lots of windows formats will not work. Use something like ‘Cool Edit’ or similar to tailor the wav files to a suitable format.

The ideal sample rate of the WAV file is 16Khz-Mono and the maximum should be 22Khz. Any higher sample rate will extremely slow down the system. Sample rates below 12Khz, the PWM will cause aliasing (filtering is a bare minimum).

If you only hear noise or random snippets of sound remember, the Speed and Capacity of the memory card are important, most 2Gb cards should be fine, 64mb cards fail all but the most-simple sounds.

Library Function: `file_PlayWAV`

Syntax: `cmd(word)`, `filename.WAV(string)`

Commands	Description
cmd	0x000B
filename.4XE	Name of the wav file to be opened and played. Filename must be 8.3 format. char0, char1, char2, ..., charN, NULL

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: If there are no errors, returns number of blocks to play (1 to 32767)

  If errors occurred, the following is returned

  6 : can’t play this rate

  5 : no data chunk found in first sector

  4 : no format data

  3 : no wave chunk signature

  2 : bad wave file format

  1 : file not found.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), char0, char1, char2, char3, char4, char5, char6, char7, char8,
// char9, char10

0x00, 0x0B, 0x43, 0x48, 0x49, 0x4D, 0x45, 0x53, 0x2E, 0x57, 0x41, 0x56, 0x00

// This will open the “CHIMES.WAV” file (0x43, 0x48, 0x49, 0x4D, 0x45, 0x53, 0x2E, 0x57,
// 0x41, 0x56) and play it, the string is appended with a Null (0x00)

// The response will be 0x06, 0x00, 0x1E assuming the command was successful, and it
// returned there are 30 blocks (0x00, 0x1E) of the WAV file to play.
```

See also

The “File Mount“ command, to initially mount the file system.
All ‘Sound Control Commands’, section 5.7

5.6.35. To Load String for 4XE/4FN File

Load the Memory space with the string to be used by the "File Call Function", "File Run" and "File Execute" commands as an argument.

The Memory Space for the "Read String for 4XE/4FN File" command or "Load String for 4XE/4FN File" command is pre-allocated memory, 512 bytes in size. It doesn't need to be released.

Library Function: `writeString`

Syntax: `cmd(word)`, `handle(word)`, `string(string)`

Commands	Description
cmd	0x0021
handle	A string pointer to the memory area where the string is to be loaded. The first string would start with handle = 0, next one would use the handle = string pointer returned from the execution of the Write string earlier.
string	A Null terminated string which is to be passed to the Child (4XE or 4FN) program.

Returns: `acknowledge(byte)`, `pointer(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *pointer*: Returns a pointer to the memory allocation where the string has been loaded.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), char0, char1, char2, char3, char4, NULL

0x00, 0x21, 0x11, 0xA9, 0x48, 0x65, 0x6C, 0x6C, 0x6F, 0x00

// This will Load the String "Hello" (0x48, 0x65, 0x6C, 0x6C, 0x6F) which has been NULL
// terminated (0x00) into the designated string pointer location 4521 (0x11, 0xA9)

// The response will be 0x06, 0x01, 0x0E assuming the command was successful and the
// pointer where the string was loaded was 4522 (0x11, 0xAA)
```

See also

The "File Mount" command, to initially mount the file system.
"File Call Function", "File Run" and "File Execute" commands to invoke a loaded function
"Read String for 4XE/4FN File" to read the string from the invoked function

5.6.36. Read String for 4XE/4FN File

Allocate and read the string from the Memory space returned by **File Call Function**, **File Run** and **File Execute** functions as an argument.

The Memory Space for the "Read String for 4XE/4FN File" and "Load String for 4XE/4FN File" commands is pre-allocated memory, 512 bytes in size. It doesn't need to be cleared.

> **Note**
>
> You have to write to a string first using the "Load String for 4XE/4FN File" command to get a handle, you pass that to the program, the handle will be used by the child program to write to what it intends to return, then you use the same handle to read what is being returned by the child program.

If you only have one string then you can write anything to it, if you have 2 and the first one is written to by the child program then the initial write must be longer than the maximum returned string.

See the examples listed under the "File Run", "File Execute" and "File Call Function" commands.

Library Function: `readString`

Syntax: `cmd(word)`, `handle(word)`

Commands	Description
cmd	0x0022
handle	A string pointer to the memory area where the string is returned from the child (4FN or 4XE) program. The first string would start with handle = 0, next one would use the handle = string pointer returned from the execution of the Write string earlier.

Returns: `acknowledge(byte)`, `string(string)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *string*: A string without NULL terminator.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB)

0x00, 0x22, 0x01, 0x0E

// This will read the string from the memory space with the handle 270 (0x01, 0x0E),
// and return the string from that memory space, without the NULL terminator.

// The response will be 0x06, 0x49, 0x20, 0x68, 0x61, 0x76, 0x65, 0x20, 0x72, 0x65,
// 0x74, 0x75, 0x72, 0x6E, 0x65, 0x64 assuming the command was successful and the string
// that was returned was “I have returned” (0x49, 0x20, 0x68, 0x61, 0x76, 0x65, 0x20, 0x72,
// 0x65, 0x74, 0x75, 0x72, 0x6E, 0x65, 0x64)
```

See also

The “File Mount” command, to initially mount the file system.
File Call Function”, “File Run” and “File Execute” commands to invoke a loaded function
“Load String for 4XE/4FN File” to read the string from the invoked function

5.7. Sound Control Commands

The following is a summary of the commands available to be used for Sound Control:

- Sound Volume
- Sound Pitch
- Sound Buffer
- Sound Stop
- Sound Pause
- Sound Continue
- Sound Playing

> **Note**
>
> All these commands are used in conjunction with Play WAV file command.

5.7.1. Sound Volume

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.

Library Function: `snd_Volume`

Syntax: `cmd(word)` , `level(word)`

Commands	Description
cmd	0xFE35
level	Sound playback volume level. 0 - 127

Returns: 0x06: ACK byte if successful,anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), level(MSB), level(LSB)

0xFE, 0x35, 0x00, 0x64

// This will set the volume to be 100 (0x00, 0x64) out of the possible 127

// The response will be 0x06 if the command was successful
```

See also

The “File Mount” command, to initially mount the file system.
“Play WAV File” command, to open the WAV file to be played

5.7.2. Sound Pitch

Sets the samples playback rate to a different frequency. Setting pitch to zero restores the original sample rate.

Library Function: `snd_Pitch`

Syntax: `cmd(word)` , `pitch(word)`

Commands	Description
cmd	0xFE34
pitch	Sample's playback rate. Minimum is 4KHz. Range is, 4000 – 65535.

Returns: `acknowledge(byte)` , `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *string*: Returns sample's original sample rate.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), pitch(MSB), pitch(LSB)

0xFE, 0x34, 0x50, 0x14

// This will set the pitch to be 20500 (0x40, 0x14) out of the possible 65535

// The response will be 0x06 if the command was successful
```

See also

The “File Mount” command, to initially mount the file system.
“Play WAV File” command, to open the WAV file to be played

5.7.3. Sound Buffer

Specify the 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 command is for control of a wav buffer, see the "Play WAV File" command

Library Function: `snd_BufSize`

Syntax: `cmd(word)`, `buffersize(word)`

Commands	Description
cmd	0xFE33
buffersize	Specifies the buffer size. 0 = 1024 bytes (default) 1 = 2048 bytes 2 = 4096 bytes

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), buffersize(MSB), buffersize(LSB)

0xFE, 0x33, 0x00, 0x01

// This will set the sound buffer size to be 2048 bytes (0x00, 0x01)

// The response will be 0x06 if the command was successful
```

See also

The "File Mount" command, to initially mount the file system.
"Play WAV File" command, to open the WAV file to be played

5.7.4. Sound Stop

Stop any sound that is currently playing, releasing buffers and closing any open WAV file.

This command is for control of a wav buffer, see the "Play WAV File" command

Library Function: `snd_Stop`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFE32

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFE, 0x32

// This will stop any currently playing sound

// The response will be 0x06 if the command was successful
```

See also

The "File Mount" command, to initially mount the file system.
"Play WAV File" command, to open the WAV file to be played

5.7.5. Sound Pause

Pause any sound that is currently playing.

This command is for control of a wav buffer, see the "Play WAV File" command

Library Function: `snd_Pause`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFE31

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFE, 0x31

// This will pause any currently playing sound

// The response will be 0x06 if the command was successful
```

See also

The "File Mount" command, to initially mount the file system.
"Play WAV File" command, to open the WAV file to be played

5.7.6. Sound Continue

Resume any sound that is currently paused by the "Sound Pause" command. This command is for control of a wav buffer, see the "Play WAV File" command.

Library Function: `snd_Continue`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFE30

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFE, 0x30

// This will continue any currently paused sound

// The response will be 0x06 if the command was successful
```

See also

The "File Mount" command, to initially mount the file system.
"Play WAV File" command, to open the WAV file to be played

5.7.7. Sound Playing

Returns 0 if sound has finished playing, else return number of 512 byte blocks to go. This command is for control of a wav buffer, see the "Play WAV File" command.

Library Function: `snd_Playing`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFE2F

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Number of 512 byte blocks to go.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFE, 0x2F

// This command will return the number of 512 byte blocks remaining on the currently
// playing sound file.

// The response will be 0x06, 0x26, 0x2A assuming the command was successful and the
// currently playing WAV file had 9770 blocks (0x26, 0x2A) of 512 bytes remaining to play.
```

See also

The "File Mount" command, to initially mount the file system.
"Play WAV File" command, to open the WAV file to be played

5.8. Touch Screen Commands

The following is a summary of the commands available to be used for Touch Screens:

- Touch Detect Region
- Touch Set
- Touch Get

5.8.1. Touch Detect Region

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" command.

Library Function: `touch_DetectRegion`

Syntax: `cmd(word)`, `x1(word)`, `y1(word)`, `x2(word)`, `y2(word)`

Commands	Description
cmd	0xFE6A
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: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), line(MSB), line(LSB), column(MSB), column(LSB)

0xFE, 0x6A, 0x00, 0x00, 0x00, 0x00, 0x00, 0x64, 0x00, 0x64

// This will set a touch region between X1=0 (0x00, 0x00), Y1=0 (0x00, 0x00) and X2=100
// (0x00, 0x64), Y2=100 (0x00, 0x64).

// The response will be 0x06 if the command was successful.
```

5.8.2. Touch Set

Sets various Touch Screen related parameters.

Library Function: `touch_Set`

Syntax: `cmd(word)`, `mode(word)`

Commands	Description
cmd	0xFE69
mode	**mode = 0:** Enables and initialises Touch Screen hardware. **mode = 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:** This will reset the current active region to default which is the full screen area

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB)

0xFE, 0x69, 0x00, 0x00

// This will enable and initialise the touch screen hardware, Mode = 0 (0x00, 0x00)

// The response will be 0x06 assuming the command was successful
```

5.8.3. Touch Get

Returns various Touch Screen parameters to caller, based on the touch detect region on the screen set by the "Touch Detect Region" command.

Library Function: `touch_Get`

Syntax: `cmd(word)` , `mode(word)`

Commands	Description
cmd	0xFE68
mode	mode = 0 : Get Status mode = 1 : Get X coordinates mode = 2 : Get Y coordinates.

Returns: `acknowledge(byte)` , `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *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

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB)

0xFE, 0x68, 0x00, 0x01

// This will get the current X coordinate of where the users finger is on the touch screen,
// in the touch region, using Mode = 1 (0x00, 0x01)

// The response will be 0x06, 0x00, 0x47 assuming the command was successful and the users
// finger was located at X=71 (0x00, 0x47)
```

5.9. Image Control Commands

The following is a summary of the commands available to be used for Image Control:

- Image Set Position
- Image Enable
- Image Disable
- Image Darken
- Image Lighten
- Set Image Parameters
- Get Image Parameters
- Show Image
- Set Image Attributes
- Clear Image Attributes
- Image Touched
- Blit Com to Display

Note

All these commands are used in conjunction with the file Load Image Control command.

5.9.1. Image Set Position

This function requires that an image control has been created with the "Load Image Control" command.

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 the "Show Image" command is called, the image will not be shown.

Library Function: `img_SetPosition`

Syntax: `cmd(word)`, `handle(word)`, `index(word)`, `xpos(word)`, `ypos(word)`

Commands	Description
cmd	0xFE8A
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: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If the operation successful, else **0**: if the attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), index(MSB), index(LSB), xpos(MSB),
// xpos(LSB), ypos(MSB), ypos(LSB)

0xFE, 0x8A, 0x11, 0xB3, 0x00, 0x01, 0x00, 0x19, 0x00, 0x0A

// This will set the position of the top left corner of the image to be displayed at
// X=25 (0x00, 0x19), Y=10 (0x00, 0x0A), where the image has a file handle of 4531 (0x11,
0xB3)
// and the index of the required image in that file is 1 (0x00, 0x01).

// The response will be 0x06, 0x00, 0x01 assuming the command was successful (0x06) and the
// operation was successful (0x00, 0x01)
```

5.9.2. Image Enable

This command requires that an image control has been created with the "Load Image Control" command.

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 the "Show Image" command 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.

Library Function: `img_Enable`

Syntax: `cmd(word)`, `handle(word)`, `index(word)`

Commands	Description
cmd	0xFE89
handle	Pointer to the Image List.
index	Index of the images in the list.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If the operation successful, else **0**: if the attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), index(MSB), index(LSB)

0xFE, 0x89, 0x11, 0xB3, 0x00, 0x01

// This will enable the image with index = 1 from the image which has a handle of 4531
// (0x11, 0xB3)

// The response will be 0x06, 0x00, 0x01 assuming the command was successful (0x06) and
// the operation was successful (0x00, 0x01)
```

5.9.3. Image Disable

This function requires that an image control has been created with the "Load Image Control" command. 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 the "Show Image" command 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.

Library Function: `img_Disable`

Syntax: `cmd(word)`, `handle(word)`, `index(word)`

Commands	Description
cmd	0xFE88
handle	Pointer to the Image List.
index	Index of the images in the list.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If the operation successful, else **0**: if the attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), index(MSB), index(LSB)

0xFE, 0x88, 0x11, 0xB3, 0x00, 0x02

// This will disable the image with index = 2 from the image which has a handle of
// 4531 (0x11, 0xB3)

// The response will be 0x06, 0x00, 0x01 assuming the command was successful (0x06) and
// the operation was successful (0x00, 0x01)
```

5.9.4. Image Darken

This function requires that an image control has been created with the "Load Image Control" command.

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 the "Show Image" command is called the control will take effect. To darken all the images in the list at the same time set index to -1.

> **Note**
>
> This feature will take effect one time only and when the "Show Image" command is called again the darkened image will revert back to normal.

Library Function: `img_Darken`

Syntax: `cmd(word)`, `handle(word)`, `index(word)`

Commands	Description
cmd	0xFE87
handle	Pointer to the Image List.
index	Index of the images in the list.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If the operation successful, else **0**: if the attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), index(MSB), index(LSB)

0xFE, 0x87, 0x11, 0xB3, 0xFF, 0xFF

// This will darken all of the images in the list that will next be shown by using the
// index = -1 (0xFF, 0xFF), using the image file which has a handle of 4531 (0x11, 0xB3)

// The response will be 0x06, 0x00, 0x01 assuming the command was successful (0x06) and
// the operation was successful (0x00, 0x01)
```

5.9.5. Image Lighten

This function requires that an image control has been created with the "Load Image Control" command.

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 the "Show Image" command is called the control will take effect. To lighten all the images in the list at the same time set index to -1.

> **Note**
>
> This feature will take effect one time only and when the "Show Image" command is called again the lightened image will revert back to normal.

Library Function: `img_Lighten`

Syntax: `cmd(word)`, `handle(word)`, `index(word)`

Commands	Description
cmd	0xFE86
handle	Pointer to the Image List.
index	Index of the images in the list.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If the operation successful, else **0**: if the attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), index(MSB), index(LSB)

0xFE, 0x86, 0x11, 0xB3, 0x00, 0x01

// This will lighten the images in the list that has the index = 1 (0x00, 0x01),
// using the image file which has a handle of 4531 (0x11, 0xB3)

// The response will be 0x06, 0x00, 0x01 assuming the command was successful (0x06)
// and the operation was successful (0x00, 0x01)
```

5.9.6. Set Image Parameters

This function requires that an image control has been created with the "Load Image Control" command.

Set image parameters in an image entry.

> **Note**
>
> The "Show Image" command will now show an 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.

Library Function: `img_SetWord`

Syntax: `cmd(word)`, `handle(word)`, `index(word)`, `offset(word)`, `value(word)`

Commands	Description
cmd	0xFE85
handle	Pointer to the Image List.
index	Index of the images in the list.
offset	Offset of the required word in the image entry. 2 - IMAGE_XPOS // WORD image location X 3 - IMAGE_YPOS // WORD image location Y 6 - IMAGE_FLAGS // WORD image flags 7 - IMAGE_DELAY // WORD inter frame delay 9 - IMAGE_INDEX // WORD current frame **Note**: Not all Constants are listed as some are Read Only.
value	The word to be written to the entry.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If the operation successful, else **0**: if the attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), index(MSB), index(LSB), offset(MSB),
// offset(LSB), value(MSB), value(LSB)

0xFE, 0x85, 0x0D, 0xE4, 0x00, 0x01, 0x00, 0x04, 0x00, 0x64

// This will set the IMAGE_WIDTH parameter (0x00, 0x04) of the image with a handle of
// 3556 (0x0D, 0xE4) and image index of 1 (0x00, 0x01) to have the value of 100 (0x00, 0x64)

// The response will be 0x06, 0x00, 0x01 assuming the command was successful (0x06) and
// the operation was successful (0x00, 0x01)
```

5.9.7. Get Image Parameters

This function requires that an image control has been created with the "Load Image Control" command.

Returns the image parameters in an image entry.

Library Function: `img_GetWord`

Syntax: `cmd(word)`, `handle(word)`, `index(word)`, `offset(word)`

Commands	Description
cmd	0xFE85
handle	Pointer to the Image List.
index	Index of the images in the list.
offset	Offset of the required word in the image entry. 2 - IMAGE_XPOS // WORD image location X 3 - IMAGE_YPOS // WORD image location Y 4 - IMAGE_WIDTH // WORD image width 5 - IMAGE_HEIGHT // WORD image height 6 - IMAGE_FLAGS // WORD image flags 7 - IMAGE_DELAY // WORD inter frame delay 8 - IMAGE_FRAMES // WORD number of frames 9 - IMAGE_INDEX // WORD current frame **Note**: Not all Constants are listed as some are Read Only.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: The word to be written to the entry.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), index(MSB), index(LSB), offset(MSB),
// offset(LSB)

0xFE, 0x84, 0x0D, 0xE4, 0x00, 0x06, 0x00, 0x05

// This will get the current IMAGE_HEIGHT (0x00, 0x05) value from the image, which has a
// handle of 3556 (0x0D, 0xE4), and index of 6 (0x00, 0x05)

// The response will be 0x06, 0x00, 0x49 assuming the command was successful and the
// Image Height was reported to be 73 (0x00, 0x49).
```

5.9.8. Show Image

This function requires that an image control has been created with the "Load Image Control" command.

Enable the displaying of the image entry in the image control.

Returns a non-zero value if successful but return value is usually ignored.

Library Function: `img_Show`

Syntax: `cmd(word)`, `handle(word)`, `index(word)`

Commands	Description
cmd	0xFE83
handle	Pointer to the Image List.
index	Index of the images in the list.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If the operation successful, else **0**: if the attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), index(MSB), index(LSB)

0xFE, 0x83, 0x0D, 0xE4, 0x00, 0x01

// This will show the image which has a handle of 3556 (0x0D, 0xE4) and image index of
// 1 (0x00, 0x01)

// The response will be 0x06, 0x00, 0x01 assuming the command was successful and the
// image show operation was successful (return may be any non-zero value) (0x00, 0x01)
```

5.9.9. Set Image Attributes

This command 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 below).

A '1' bit in the "value" field SETS the respective bit in the IMAGE_FLAGS field of the image control entry.

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_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, default=0 for image

Library Function: `img_SetAttributes`

Syntax: `cmd(word)`, `handle(word)`, `index(word)`, `value(word)`

Commands	Description
cmd	0xFE82
handle	Pointer to the Image List.
index	Index of the images in the list.
value	Refer to the Image Attribute Flags in the description above.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: TRUE or FALSE

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), index(MSB), index(LSB), value(MSB),
// value(LSB)

0xFE, 0x82, 0x11, 0xB3, 0x00, 0x01, 0x40, 0x00

// This will set the image with handle=4531 (0x11, 0xB3) with index=1 (0x00, 0x01)
// that is next shown with the “Show Image” command to be Darker (0x40, 0x00),
// the same as using the “Image Darken” command.

// The response will be 0x00, 0x00, 0x01 assuming the command was successful and
// the image attribute was successfully set (0x00, 0x01).
```

5.9.10. Clear Image Attributes

Clear various Image Attribute Flags in an image control entry. (see image attribute flags below) Image Attribute Flags may be combined by adding the hex of two or more flags together, or with binary addition.

This function requires that an image control has been created with the "Load Image Control" command. 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_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, default=0 for image

Library Function: `img_ClearAttributes`

Syntax: `cmd(word)`, `handle(word)`, `index(word)`, `value(word)`

Commands	Description
cmd	0xFE81
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. Refer to the Image Attribute Flags in the description above.

Note

If index is set to -1, the attribute is altered in ALL of the entries in the image list.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: **1**: If the operation successful, else **0**: if the attempt failed.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), index(MSB), index(LSB), value(MSB),
// value(LSB)

0xFE, 0x81, 0x11, 0xB3, 0x00, 0x21, 0x80, 0x00

// This will clear the I_ENABLED (0x80, 0x00) attribute from the image with handle = 4531
// (0x11, 0xB3) and index = 33 (0x00, 0x21)

// The response will be 0x06, 0x00, 0x01 assuming the command was successful (0x06) and
// the attribute was successfully cleared (0x00, 0x01).
```

5.9.11. Image Touched

This command requires that an image control has been created with the "Load Image Control" command. Returns index if image touched or returns -1 image not touched. If index is passed as -1 the command tests all images and returns -1 if image not touched or returns index.

Library Function: `img_Touched`

Syntax: `cmd(word)`, `handle(word)`, `index(word)`

Commands	Description
cmd	0xFE80
handle	Pointer to the Image List.
index	Index of the images in the list.

Returns: `acknowledge(byte)`, `status(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: Returns image index if image touched.
  -1 if image not touched.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB), index(MSB), index(LSB)

0xFE, 0x80, 0x0D, 0xE4, 0x00, 0x05

// This will return if an image with handle 3556 (0x44, 0x0D) and index 5 (0x00, 0x05)
// has been touch.

// The response will be 0x06, 0x00, 0x05 assuming the command was successful and the
// image touched had the index of 5 (0x00, 0x05).
```

5.9.12. Blit Com to Display

This command will BLIT (Block Image Transfer) 16-bit pixel data from the Com port on to the screen.

Library Function: `blitComtoDisplay`

Syntax: `cmd(word)`, `x(word)`, `y(word)`, `width(word)`, `height(word)`, `data(data)`

Commands	Description
cmd	0x0023
x,y	Specifies the horizontal and vertical position of the top-left corner of the image to be displayed
width	width of the image to be displayed
height	height of the image to be displayed
data	pixel1...pixeln 16 bit pixel data to be plotted on the Display screen. 16 bit = 5bit Red, 6bit Green, 5bit Blue

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), x(MSB), x(LSB), y(MSB), y(LSB), width(MSB), width(LSB), height(MSB),
// height(LSB), pixel1, pixel2, ..., pixelN

0x00, 0x23, 0x00, 0x00, 0x00, 0x00, 0x01, 0xE0, 0x00, 0xBC, 0x31, 0x81, 0x63... etc

// This will displaying an image at X=0 (0x00, 0x00), Y=0 (0x00, 0x00) with Width = 480
// (0x01, 0xE0) and height = 188 (0x00, 0xBC)

// The response will be 0x06 assuming the command was successful
```

5.10. System Commands

The following is a summary of the commands available to be used for System:

- Memory Release
- Memory Status
- Get Display Model
- Get SPE Version
- Get PmmC Version

5.10.1. Memory Release

The 'memory release' command releases the memory space used by the 'Load Image Control' and 'file Load Function' commands.

Library Function: `mem_Free`

Syntax: `cmd(word)`, `handle(word)`

Commands	Description
cmd	0xFE5F
handle	Pointer to the memory block.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *status*: 0: If the attempt failed.

  Non-0: If the operation successful.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), handle(MSB), handle(LSB)

0xFE, 0x5F, 0x11, 0xB3

// This will release the memory utilized by the handle 4531 (0x11, 0xB3)

// The response will be 0x06, 0x00, 0x01 assuming the command was successful and
// the operation was successful.
```

5.10.2. Memory Status

Returns byte size of the largest chunk of memory available in the heap.

Library Function: `mem_Heap`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFE5E

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Returns the largest available memory chunk of the heap.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFE, 0x5E

// This will return the largest available chunk of memory in the heap

// The response will be 0x06, 0x26, 0x86 assuming the command was successful and the
// display reported back 9862 (0x26, 0x86) bytes of available memory in its largest chunk
```

5.10.3. Get Display Model

Returns the Display Model in the form of a string without Null terminator

Library Function: `sys_GetModel`

Syntax: `cmd(word)`

Commands	Description
cmd	0x001A

Returns: `acknowledge(byte)`, `model(string)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *count*: Number of characters in the model name to return
- *model*: Display Module's model name. Without NULL terminator.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0x00, 0x1A

// This will request the display to return its model name as a string of characters
// without the NULL

// The response will be 0x06, 0x00, 0x0A, 0x75, 0x4C, 0x43, 0x44, 0x2D, 0x33, 0x32,
// 0x50, 0x54, 0x55 assuming the command was successful and the display returned 10
// characters (0x00, 0x0A) and the display model was “uLCD-32PTU”
// (0x75, 0x4C, 0x43, 0x44, 0x2D, 0x33, 0x32, 0x50, 0x54, 0x55)
```

5.10.4. Get SPE Version

Returns the SPE Version installed on the module.

Library Function: `sys_GetVersion`

Syntax: `cmd(word)`

Commands	Description
cmd	0x001B

Returns: `acknowledge(byte)`, `version(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *version*: SPE Version installed on the module.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0x00, 0x1B

// This will return the version of the SPE Application loaded into the display

// The response will be 0x06, 0x01, 0x00 assuming the command was successful and
// the version of the SPE Application was 256 (0x01, 0x00)
```

5.10.5. Get PmmC Version

Returns the PmmC Version installed on the module.

Library Function: `sys_GetPmmC`

Syntax: `cmd(word)`

Commands	Description
cmd	0x001C

Returns: `acknowledge(byte)`, `version(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *version*: PmmC Version installed on the module.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0x00, 0x1C

// This will return the version of the PmmC loaded into the display

// The response will be 0x06, 0x03, 0x03 assuming the command was successful and
// the PmmC loaded was version 771 (0x03, 0x03)
```

5.10.6. Peek Memory

Returns the word contents of a specified memory address. This command would normally be used to read the contents of File and/or ImageControl handles.

Library Function: `peekM`

Syntax: `cmd(word)`, `address(word)`

Commands	Description
cmd	0x0027
address	The address to be peeked.

Returns: `acknowledge(byte)`, `contents(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *contents*: The contents of the specified memory address.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), address(MSB), address(LSB)

0x00, 0x27, 0x14, 0x3C

// This example assumed a file had been opened and the handle returned was at 0x142A.
// Offset 18 from this (0x143C) is the FILE_ATTRIBUTES word.

// The response will be 0x06, 0x00, 0x20 assuming the command was successful and the
// file had the Archive bit set.
```

5.10.7. Poke Memory

Sets the word contents of a specified memory address. This command would normally be used to alter the contents of File and/or ImageControl handles.

Library Function: `pokeM`

Syntax: `cmd(word)`, `address(word)`, `wordvalue(word)`

Commands	Description
cmd	0x0028
address	The address to be poked.
wordvalue	The data to be poked into the address

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), address(MSB), address(LSB)

0x00, 0x27, 0x14, 0x3C, 0x00, 0x00

// This example assumed a file had been opened and the handle returned was at 0x142A.
// Offset 18 from this (0x143C) is the FILE_ATTRIBUTES word.

// The response will be 0x06 assuming the command was successful. This example would
// clear the Archive bit.
```

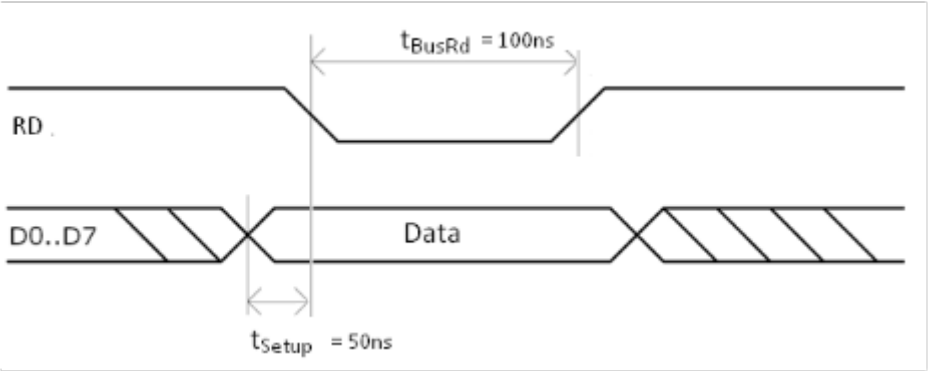
5.11. I/O Commands

The following is a summary of the commands available to be used for I/O Control:

- BUS Read8
- BUS Write8
- Pin HI
- Pin LO
- Pin Read
- Pin Set

5.11.1. BUS Read8

Returns the state of the bus as a 8bit value in to the lower byte of the assigned variable. Bus pins can be set as either INPUT or OUTPUT, using the BUS Set command.



Pin Constants	Description
PA4	BUS Pin 0, pin = 5, physical pin = 46 (Diablo16)
PA5	BUS Pin 1, pin = 6, physical pin = 49 (Diablo16)
PA6	BUS Pin 2, pin = 7, physical pin = 50 (Diablo16)
PA7	BUS Pin 3, pin = 8, physical pin = 51 (Diablo16)
PA8	BUS Pin 4, pin = 9, physical pin = 52 (Diablo16)
PA9	BUS Pin 5, pin = 10, physical pin = 53 (Diablo16)
PA10	BUS Pin 6, pin = 11, physical pin = 43 (Diablo16)
PA11	BUS Pin 7, pin = 12, physical pin = 44 (Diablo16)

- *BUS_WR* is PA2
- *BUS_RD* is PA3

Please refer to the datasheet of the display module you are using, to determine which pin on your module is *BUS_RD*.

Note

- 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.

Library Function: `bus_Read8`

Syntax: `cmd(word)`

Commands	Description
cmd	0xFF86

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Returns the state of the bus as a 8bit value.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB)

0xFF, 0x86

// This will return the value of the BUS pins

// The response could be 0x06, 0x00, 0xEC assuming the command was successful and the
// BUS has BUS pins 2, 3, 5, 6 and 7 HI (PA6, PA7, PA9, PA10 and PA11) and the rest
// LO (0x00, 0xEC) or (11101100 in Binary)
```

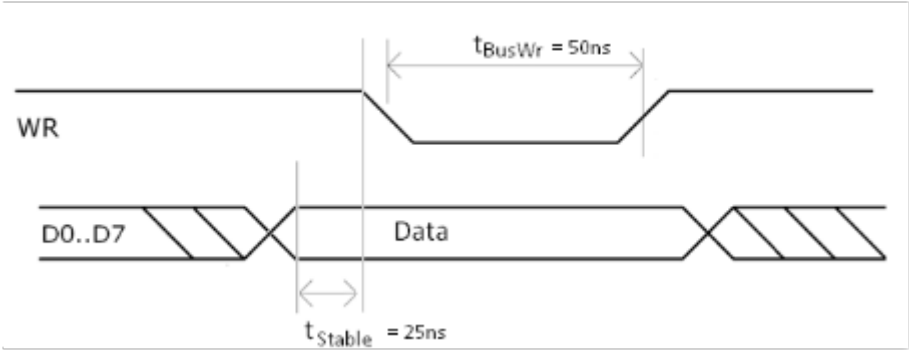
See also

Bus Set command, to determine if the pin is an INPUT or an OUTPUT

5.11.2. BUS Write8

Sets the value of the BUS pins. The BUS pins should be set as OUTPUT first, using the BUS Set command.

The lower 8 bits of arg1 are placed on the BUS, then, after a settling delay of approximately 25nsec, the *BUS_WR* pin is strobed LO for approx 50nsec then set back HI.



The upper 8 bits of arg1 are ignored.

Pin Constants	Description
PA4	BUS Pin 0, pin = 5, physical pin = 46 (Diablo16)
PA5	BUS Pin 1, pin = 6, physical pin = 49 (Diablo16)
PA6	BUS Pin 2, pin = 7, physical pin = 50 (Diablo16)
PA7	BUS Pin 3, pin = 8, physical pin = 51 (Diablo16)
PA8	BUS Pin 4, pin = 9, physical pin = 52 (Diablo16)
PA9	BUS Pin 5, pin = 10, physical pin = 53 (Diablo16)
PA10	BUS Pin 6, pin = 11, physical pin = 43 (Diablo16)
PA11	BUS Pin 7, pin = 12, physical pin = 44 (Diablo16)

- *BUS_WR* is PA2
- *BUS_RD* is PA3

Please refer to the datasheet of the display module you are using, to determine which pin on your module is *BUS_WR*.

> **Note**
>
> The *BUS_WR* pin is automatically pre-set to an output to ensure BUS write integrity.

Library Function: `bus_Write8`

Syntax: `cmd(word)`, `arg(word)`

Commands	Description
cmd	0xFF87
arg	Argument specifying the pins on the bus to output. The lower byte of the argument is placed on the 8bit wide bus. The upper byte of the argument is ignored

Returns: 0x06: ACK byte if successful, anything else implies mismatch between command and response.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), arg(MSB), arg(LSB)

0xFF, 0x87, 0x00, 0x02

// This will output HI on to BUS pin 1 (PA5) and LO on to the rest of the BUS pins
// (0x00, 0x02 is 00000010 in binary)

// The response could be 0x06 assuming the command was successful.
```

See also

Bus Set command, to determine if the pin is an INPUT or an OUTPUT

5.11.3. Pin HI

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. I/O pins should be set as OUTPUT first, using the Pin Set/Bus Set commands.

Pin constants	Description
PA0	I/O Pin 1, pin = 1, physical pin = 61 (Diablo16)
PA1	I/O Pin 2, pin = 2, physical pin = 62 (Diablo16)
PA2	I/O Pin 3, pin = 3, physical pin = 63 (Diablo16)
PA3	I/O Pin 4, pin = 4, physical pin = 64 (Diablo16)
PA4	I/O Pin 5, pin = 5, physical pin = 46 (Diablo16)
PA5	I/O Pin 6, pin = 6, physical pin = 49 (Diablo16)
PA6	I/O Pin 7, pin = 7, physical pin = 50 (Diablo16)
PA7	I/O Pin 8, pin = 8, physical pin = 51 (Diablo16)
PA8	I/O Pin 9, pin = 9, physical pin = 52 (Diablo16)
PA9	I/O Pin 10, pin = 10, physical pin = 53 (Diablo16)
PA10	I/O Pin 11, pin = 11, physical pin = 43 (Diablo16)
PA11	I/O Pin 12, pin = 12, physical pin = 44 (Diablo16)
PA12	I/O Pin 13, pin = 13, physical pin = 31 (Diablo16)
PA13	I/O Pin 14, pin = 14, physical pin = 32 (Diablo16)

Note

Constant variables available for use when using a 4D Labs Serial library.

Library Function: `pin_Hi`

Syntax: `cmd(word)` , `pin(word)`

Commands	Description
cmd	0xFF8F
pin	A value specifying the pin number.

Returns: `acknowledge(byte)` , `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Returns 1 if the pin value was a legal number.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), pin(MSB), pin(LSB)

0xFF, 0x8F, 0x00, 0x04

// This will set Pin 4 (PA3) to output HI

// The response could be 0x06, 0x00, 0x01 assuming the command was successful, and the
// pin number was legal (0x00, 0x01)
```

See also

Pin Set command, to determine if the pin is an INPUT or an OUTPUT

5.11.4. Pin LO

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. I/O pins should be set as OUTPUT first, using the Pin Set command.

Pin constants	Description
PA0	I/O Pin 1, pin = 1, physical pin = 61 (Diablo16)
PA1	I/O Pin 2, pin = 2, physical pin = 62 (Diablo16)
PA2	I/O Pin 3, pin = 3, physical pin = 63 (Diablo16)
PA3	I/O Pin 4, pin = 4, physical pin = 64 (Diablo16)
PA4	I/O Pin 5, pin = 5, physical pin = 46 (Diablo16)
PA5	I/O Pin 6, pin = 6, physical pin = 49 (Diablo16)
PA6	I/O Pin 7, pin = 7, physical pin = 50 (Diablo16)
PA7	I/O Pin 8, pin = 8, physical pin = 51 (Diablo16)
PA8	I/O Pin 9, pin = 9, physical pin = 52 (Diablo16)
PA9	I/O Pin 10, pin = 10, physical pin = 53 (Diablo16)
PA10	I/O Pin 11, pin = 11, physical pin = 43 (Diablo16)
PA11	I/O Pin 12, pin = 12, physical pin = 44 (Diablo16)
PA12	I/O Pin 13, pin = 13, physical pin = 31 (Diablo16)
PA13	I/O Pin 14, pin = 14, physical pin = 32 (Diablo16)

Note

Constant variables available for use when using a 4D Labs Serial library.

Library Function: `pin_Lo`

Syntax: `cmd(word)`, `pin(word)`

Commands	Description
cmd	0xFF8E
pin	A value specifying the pin number.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Returns 1 if the pin value was a legal number.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), pin(MSB), pin(LSB)

0xFF, 0x8E, 0x00, 0x05

// This will set Pin 5 (PA4) to output LO

// The response could be 0x06, 0x00, 0x01 assuming the command was successful,
// and the pin number was legal (0x00, 0x01)
```

See also

Pin Set command, to determine if the pin is an INPUT or an OUTPUT

5.11.5. Pin Read

Returns a "Low" (logic 0) or a "High" (logic 1) based on the value of the selected pin. I/O pins can be set as either INPUT or OUTPUT, using the Pin Set command

Pin constants	Description
PA0	I/O Pin 1, pin = 1, physical pin = 61 (Diablo16)
PA1	I/O Pin 2, pin = 2, physical pin = 62 (Diablo16)
PA2	I/O Pin 3, pin = 3, physical pin = 63 (Diablo16)
PA3	I/O Pin 4, pin = 4, physical pin = 64 (Diablo16)
PA4	I/O Pin 5, pin = 5, physical pin = 46 (Diablo16)
PA5	I/O Pin 6, pin = 6, physical pin = 49 (Diablo16)
PA6	I/O Pin 7, pin = 7, physical pin = 50 (Diablo16)
PA7	I/O Pin 8, pin = 8, physical pin = 51 (Diablo16)
PA8	I/O Pin 9, pin = 9, physical pin = 52 (Diablo16)
PA9	I/O Pin 10, pin = 10, physical pin = 53 (Diablo16)
PA10	I/O Pin 11, pin = 11, physical pin = 43 (Diablo16)
PA11	I/O Pin 12, pin = 12, physical pin = 44 (Diablo16)
PA12	I/O Pin 13, pin = 13, physical pin = 31 (Diablo16)
PA13	I/O Pin 14, pin = 14, physical pin = 32 (Diablo16)
PA14	I/O Pin 15, pin = 15, physical pin = 37 (Diablo16)
PA15	I/O Pin 16, pin = 16, physical pin = 36 (Diablo16)

Note

Constant variables available for use when using a 4D Labs Serial library.

Library Function: `pin_Read`

Syntax: `cmd(word)`, `pin(word)`

Commands	Description
cmd	0xFF8C
pin	A value specifying the pin number.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Returns a 0 or 1 depending on the state of the pin.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), pin(MSB), pin(LSB)

0xFF, 0x8C, 0x00, 0x09

// This will read the value of Pin 9 (PA8)

// The response could be 0x06, 0x00, 0x01 assuming the command was successful,
// and the I/O pin was set HI (0x00, 0x01)
```

See also

Pin Set command, to determine if the pin is an INPUT or an OUTPUT

5.11.6. Pin Set

There are pre-defined constants for pin:

Pin constants	Description	Pin constants
PA0	I/O Pin 1, pin = 1, physical pin = 61 (Diablo16)	Analog In Capable
PA1	I/O Pin 2, pin = 2, physical pin = 62 (Diablo16)	Analog In Capable
PA2	I/O Pin 3, pin = 3, physical pin = 63 (Diablo16)	Analog In Capable, also used for BUS_WR
PA3	I/O Pin 4, pin = 4, physical pin = 64 (Diablo16)	Analog In Capable, also used for BUS_RD
PA4	I/O Pin 5, pin = 5, physical pin = 46 (Diablo16)	
PA5	I/O Pin 6, pin = 6, physical pin = 49 (Diablo16)	
PA6	I/O Pin 7, pin = 7, physical pin = 50 (Diablo16)	
PA7	I/O Pin 8, pin = 8, physical pin = 51 (Diablo16)	
PA8	I/O Pin 9, pin = 9, physical pin = 52 (Diablo16)	
PA9	I/O Pin 10, pin = 10, physical pin = 53 (Diablo16)	
PA10	I/O Pin 11, pin = 11, physical pin = 43 (Diablo16)	
PA11	I/O Pin 12, pin = 12, physical pin = 44 (Diablo16)	
PA12	I/O Pin 13, pin = 13, physical pin = 31 (Diablo16)	
PA13	I/O Pin 14, pin = 14, physical pin = 32 (Diablo16)	
PA14	I/O Pin 15, pin = 15, physical pin = 37 (Diablo16)	Digital Input Only
PA15	I/O Pin 16, pin = 16, physical pin = 36 (Diablo16)	Digital Input Only
AUDIO_ENABLE	Amplifier Chip control pin, pin = 17, physical pin = 45 (Diablo16)	Used internally. Permanently set as Digital Output

These are pre-defined constants for **mode**, and the **pins** they are compatible with.

4D Pin Name (Predfined)		PA0	PA1	PA2	PA3	PA4	PA5	PA6	PA7	PA8	PA9	PA10	PA11	PA12	PA13	PA14	PA15
DIABLO16 Pin Number		pin 61	pin 62	pin 63	pin 64	pin 46	pin 49	pin 50	pin 51	pin 52	pin 53	pin 43	pin 44	pin 31	pin 32	pin 37	pin 36
H1 Pin Number		pin 1	pin 3	pin 5	pin 7	pin29	pin 27	pin 25	pin 23	pin 21	pin 19	pin 8	pin 6	pin 28	pin 30	pin 24	pin 26
Pin Mode (Predefined)	mode #	Generic PIN I/O Legal Settings															
PIN_INP	0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PIN_INP_HI	1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PIN_INP_LO	2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PIN_OUT	3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PIN_OUT_OD	4	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PIN_AN	5	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
PIN_ANAVG	6	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Library Function: `pin_Set`

Syntax: `cmd(word)`, `mode(word)`, `pin(word)`

Commands	Description
cmd	0xFF90
mode	A value specifying the pin mode.
pin	A value specifying the pin number.

Returns: `acknowledge(byte)`, `value(word)`

- *acknowledge*: 0x06: ACK byte if successful, anything else implies mismatch between command and response.
- *value*: Returns 1 if the pin value was a legal number.

Example

```
// Byte Stream:
// cmd(MSB), cmd(LSB), mode(MSB), mode(LSB), pin(MSB), pin(LSB)

0xFF, 0x90, 0x00, 0x05, 0x00, 0x04

// This will set Pin 4 (PA3) as an Analog Input (Mode 5)

// The response could be 0x06, 0x00, 0x01 assuming the command was successful,
// and the I/O pin specified was a valid pin number (0x00, 0x01)
```

6. Revision History

Document Revision

Revision	Date	Description
1.0	21/03/2014	First Release
1.1	04/05/2014	Fixed FONT references which were incorrectly copied from Picaso
1.2	07/05/2014	Updated image in Section 2.2
1.3	01/10/2014	Fixed typo in putstr function reference (was putStr)
1.4	30/10/2014	Fixed a few typos regarding Contrast. All DIABLO-16 modules are 0-15
1.5	22/12/2014	Added information for file_LoadImageControl. Updated control block size in file_Mount. Added information relating to Set Font and uSD based fonts. Added note about restriction of clipping command. Added information about the use of TRANSPARENCY.
1.6	29/06/2015	Added max write size to “File Write” command. Fixed FontIDs for deja fonts
2.0	01/05/2017	Updated formatting and contents
2.1	21/03/2019	Updated formatting
2.2	30/12/2023	Modified for web-based documentation
2.3	19/03/2024	Updated formatting for resource centre redesign
2.4	05/06/2024	Add table class, arrange code example and replace images.

7. Legal Notice

7.1. Proprietary Information

The information contained in this document is the property of 4D Systems Pty. Ltd. and may be the subject of patents pending or granted, and must not be copied or disclosed without prior written permission. 4D Systems 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 Systems products and services is continuous and published information may not be up to date. It is important to check the current position with 4D Systems. 4D Systems 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.

7.2. Disclaimer of Warranties & Limitations of Liabilities

4D Systems 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 Systems range of products, however the quality may vary.

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

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

Use of 4D Systems' 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 Systems from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any 4D Systems intellectual property rights.