

**Digole Serial:UART/I2C/SPI Character/Graphic LCD/OLED Display Module  
Programmer Manual**

(last updated: December 8, 2018)

This manual may be modified without notice

For firmware V4.2

[Learn and try our displays online](#)

<b>Upgrade able firmware</b> .....	<b>5</b>
<b>Set up the communication mode</b> .....	<b>5</b>
<b>Brief of Commands</b> .....	<b>6</b>
<b>Escape commands</b> .....	<b>6</b>
<b>Legend:</b> .....	<b>8</b>
Deal with pixels 255 to 511 .....	8
<b>Flash Memory Map</b> .....	<b>8</b>
<b>Port Connection</b> .....	<b>9</b>
<b>Command summery</b> .....	<b>11</b>
<b>256 color code</b> .....	<b>11</b>
<b>Display characters</b> .....	<b>11</b>
Display a string (TT.....\x00).....	11
Move current position(TPxy) .....	11
Enhanced move current position(ETPxy) .....	12
Move position to last(ETB).....	12
Move position offset(ETOxy).....	12
TextCursor(CSd) .....	12
Text Alignment(ALIGNd).....	12
<b>Draw graphics</b> .....	<b>13</b>
Set current graphic position(GPxy) .....	13
Draw a pixel(DPxy) .....	13
Draw line(LNx1y1x2y2).....	13
Draw line to(LNxy) .....	13
Draw rectangle(DRx1y1x2y2).....	13
Draw filled rectangle(FRx1y1x2y2) .....	13
Draw circle(CCxyrf) .....	14
Draw image.....	14
Move area on the screen(MAxywhOxOy) .....	14
Video Box (VIDEOxywh\x00/\x01....data).....	15
<b>Drawing Decoration</b> .....	<b>16</b>

---

Clear screen(CL).....	16
Set background color(BGC) .....	16
Set foreground color(SCc,ESCrGb).....	16
Set line pattern(SLPd).....	16
Set draw direction(SDd) .....	16
Set draw mode(DMd) .....	16
Set output/draw window(DWWINxywh) .....	17
Reset draw window(RSTDW) .....	17
Clear draw window(WINCL).....	17
Set image's background transparent ("TRANS 0/1") .....	17
<b>For Mono display only -----</b>	<b>18</b>
Refresh screen instantly(FS0/1) .....	18
Set screen Normal/Inverse(INV0/1) .....	18
<b>Fonts -----</b>	<b>19</b>
Change current font(SFd) .....	19
Download standard user font(SUFnL...d...) .....	19
Download user font to flash chip(FLMWRal...d...) .....	19
Use user font in flash chip(SFFa) .....	20
<b>Command Set -----</b>	<b>20</b>
What is command set? .....	20
Write command set to flash(FLMWRal...d...) .....	20
Run command set(FLMCSa) .....	20
<b>Read data from communication port-----</b>	<b>20</b>
<b>Use EEPROM -----</b>	<b>21</b>
Write data to EEPROM(WREPal...d...) .....	21
Read data from EEPROM(RDEPal) .....	21
<b>Use Flash -----</b>	<b>21</b>
Use flash in MCU .....	21
Use flash in flash chip .....	21
Write data to flash(FLMWRal...d...).....	21

Run command set(FLMCSa) .....	22
Read data in flash chip(FLMRDa) .....	22
Erase flash memory in flash chip(FLMERa) .....	23
<b>Touch Panel</b> .....	<b>24</b>
Calibrate touch screen(TUCHC) .....	24
Read touched coordinate(RPNXYW) .....	24
Read a click event(RPNXYC) .....	24
Read touch panel instantly(RPNXYI), check screen pressed .....	24
Read voltage(RDBAT) .....	24
Read analog(RDAUX) .....	24
Read temperature(RDTMP) .....	25
<b>Power management</b> .....	<b>25</b>
Backlight brightness(BLd) .....	25
Turn screen on/off(SOOd) .....	25
Turn MCU off(DNMCU) .....	25
Turn module off(DNALL) .....	25
<b>Setting and configuration</b> .....	<b>26</b>
Start screen(welcome screen or splash screen) .....	26
Enable/disable start screen(DSSd) .....	26
Configuration show on/off(DCd) .....	26
Download start screen to module(SSSI...d...) .....	26
Change I2C address(SI2CAa) .....	26
Set SPI mode(SPIMD0~3) .....	27
Change UART baud(SBrate) .....	27
Config universal character LCD adapter(STCRcr\x80\xC0\x94xD4) .....	27
Config universal graphic LCD adapter(SLCDx...) .....	27
Adjust LCD contrast(CTx) .....	28
<b>Others</b> .....	<b>28</b>
Delay a period(DLYx) .....	28

## Upgrade able firmware

The firmware on most of our display modules are upgrade able or custom able, please visit the firmware page for details:

[www.digole.com/fw](http://www.digole.com/fw)

## Set up the communication mode

There are 3 different communication modes on all products: UART, I2C and SPI, what you need is just use solder to short the I2C/SPI jumper on adapter and make it works at I2C or SPI, if both jumpers are open, it works at UART, you can find a similar jumper like this: on board.



### PROTOCOLS:

- UART : 8-N-1, 8bits, No parity bit, 1 stop bit.
- I2C: Slave Mode, 7-bit address, default address is Hex:27, change able. This mode may give you a headache due to more signal options in I2C, but we make it works as standard, you just need setup your I2C on master controller as Standard Master Mode.
- SPI: 8-bits, MSB first, data on raise edge of SCK sampled; this is Standard setting on SPI too.

## Brief of Commands

<b>TEXT</b>	Position adjust: - "TP" set text position, - "ETP" enhanced set text position, - "ETO" set text offset, - "ETB" back to last text position Draw Text: - "TT" display text, - "TRT" text return - "ALIGN" text alignment	Appearance adjust: - "SF" set font, - "SFF" set font in flash, - "SUF" save user font, - "SC" set drawing color (8bit format), - "ESC" enhanced set drawing color (262K), - "SD" set direction, - "CS" cursor on/off, - "DM" drawing mode	- "BGC" set background color, - "DWWIN" set drawing window, - "RSTDW" reset drawing window, - "WINCL" clear drawing window use background
<b>Graphic</b>	Position adjust: - "GP" set graphic position, Draw functions: - "DP" draw pixel, - "LN" draw line, - "LT" draw line to, - "DR" draw rectangle, - "FR" draw filled rectangle, - "CC" draw circle, - "MA" move an area, - "DIM" display mono image, - "EDIM1", "EDIM2", "EDIM3" display color image (256, 65K 262K color) - "VIDEO" display a video stream	Appearance adjust: - "SD" set direction, - "SC" set color, - "ESC" enhanced set color, - "SLP" set line pattern, - "DM" drawing mode - "TRANS" set Graph transparent or not	
<b>Communication</b>	"SI2CA"-Set I2C address, "SB"-set UART baud rate, "DC"-display module configuration		
<b>Power Manage</b>	"BL"-adjust backlight, "DNALL"-put module in sleep, "DNMCU"-put MCU sleep, "SOO"-turn screen on/off		
<b>Screen</b>	"SSS"-Save start screen, "DSS"-display start screen, "SOO"-turn screen on/off, "MCD"-send command to screen, "MDT"-send data to screen, "CL"-clear screen use background color, "CT"-set contrast, "SLCD"-config mono LCD (ST7920, ST7565,KS0108), "VIDEO"- Write Video data direct to screen, V4.0V		
<b>Touch Panel</b>	"RPNXY"[I/W/C]-read x,y, I=instant, W=wait touched, C=Click, "TUCHC"- calibrate touch panel, "RDBAT"-read battery voltage, "RDAUX"-read aux pin, "RDTMP"-read chip temperature,		
<b>Flash Memory</b>	"FLMER"-flash erase, "FLMRD"-flash read, "FLMWR"-flash write, "FLMCS"-run command set in flash,"SFF"-set font in flash		
<b>EEPROM</b>	"WREP"-write data to EEPROM, "RDEP"-read data from EEPROM		
<b>Output Window</b>	"DWWIN"-set output window,"WINCL"-clean the output window,"RSTDW"-reset the output window to full screen		
<b>Other</b>	"DLY"-delay a period		

## Escape commands

**Note: user can choose to use letter commands or escape commands or combine them together, all ESC commands are 2 bytes, first byte must be value of 27 (ESC) and second byte is the index of command. As the result of our test: by using ESCAPE commands can accelerate the average processing speed up to 3%.**






e.g.: Letter command: "TT" = ESC command: 27, 01, both are 2 bytes, but for longer letter commands, it will also save some program space for user.

here is the cross reference, the number from TT is 1, and increased 1 after each commands:

"TT" (1), "ETB" (2), "ETP" (3), "ETO" (4), "ESC" (5), "SI2CA" (6), "SC" (7), "SB" (8), "SD" (9), "SF" (10), "SSS" (11), "SUF" (12), "SOO" (13), "SLP" (14), "FR" (15), "LN" (16), "LT" (17), "DC" (18), "DIM" (19), "DP" (20), "DR" (21), "DSS" (22), "DOUT" (23), "TP" (24), "BL" (25), "TRT" (26), "GP" (27), "CL" (28), "CC" (29), "CS" (30),

"CT" (31), "MA" (32), "MCD" (33), "MDT" (34), "DM" (35), "EDIM" (36), "FS" (37), "WREP" (38), "RDEP" (39),  
"INV" (40),  
"DNALL" (41), "DNMCU" (42), "SLCD" (43), "RPNXY" (44), "TUCHC" (45), "RDBAT" (46), "RDAUX" (47),  
"RDTMP" (48), "FLMER" (49), "FLMRD" (59),  
"FLMWR" (51), "FLMCS" (52), "BGC" (53), "DWWIN" (54), "RSTDW" (55), "WINCL" (56), "SPIMD" (57), "FTOB" (58),  
"DLY" (59), "TRANS" (60),  
"VIDEO" (61), "ALIGN" (62)

## Legend:

	-----	Text (character) Display
	-----	Monochrome Graphic Display
	-----	Color Graphic Display
	-----	All version of Firmware
	-----	Special version of Firmware

## Deal with pixels 255 to 511

How do we recognize value from 0 to 511 used in position and length of pixel (x,y,w,h,r)?

In most small screen, the screen size usually less than 255, one byte of data can handle such screen, but in larger screen, the size may exceed 255, in order to make your code compatible for all size of screen, we use this technical: if the value is <255, just send one byte of the value, if >=255, send first byte of 255 then follow the rest of value, eg.: for value 120, just send one byte of value of 120, if the value is 310, send 255 first, then value of 55 as second byte, the value 255+55=310.

sample routine in C:

```
void writePosition(int p)
{
    if(p>255)
    {
        write(255);    //write a byte to COM port
        p-=255;
    }
    write(p);
}
```

## Flash Memory Map

**Flash memory size:** 2MB~16MB if flash chip installed, 16KB if none flash chip installed.

**Usage:** 5 block of 65KB from address 0, used for welcome screen and 4 user fonts if flash chip installed. otherwise: 0~2047B for welcome screen, and 4x3584B for 4 user fonts.

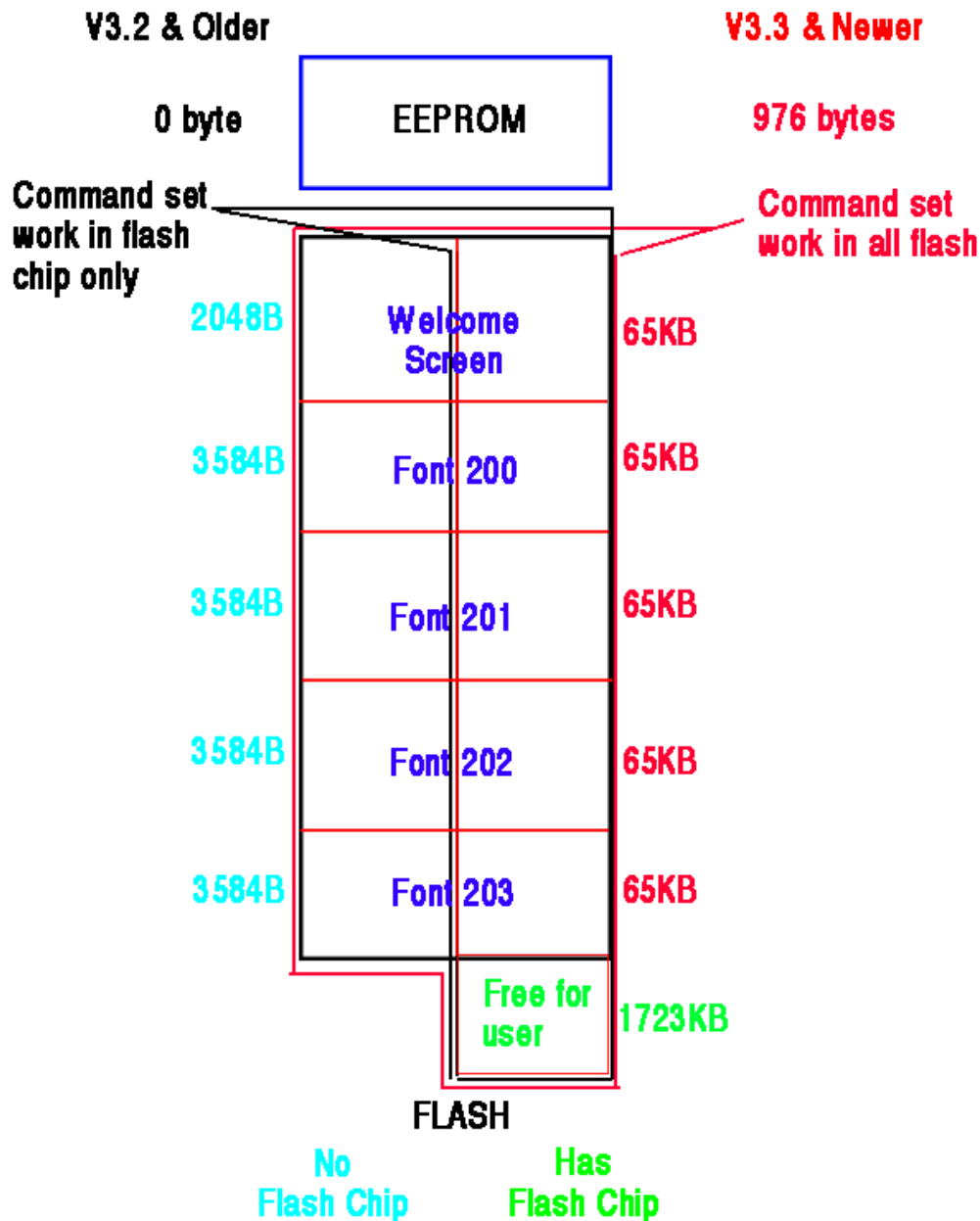
**User fonts:** unlimited user fonts in flash chip, or 4 user fonts in 16KB flash.

**Command sets:** unlimited command set in flash chip, V3.3: unlimited command set in 16KB flash.

**Data protection:** data in 16KB flash can't be read out, but data in flash chip can.

The right portion of memory map bellow is for example of 2MB flash chip.



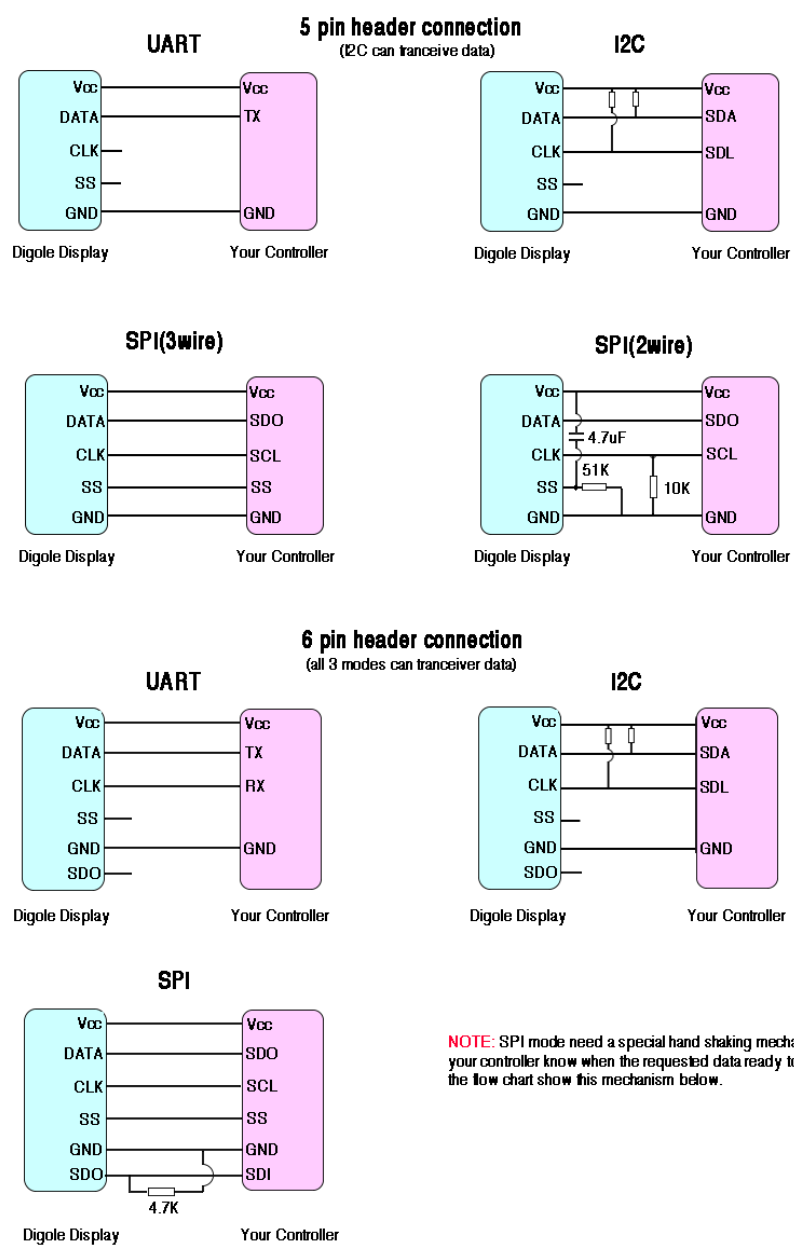


## Port Connection

In the 5 pin header modules, only I2C mode can transmit and receive data, other mode only can transmit data to display module, if you need read data from EEPROM(V3.3), you only can use I2C mode to do that. All modes can transeive data in 6 pin header modules

On the 2 wire SPI mode, if the master circuit was setting slow, the C:4.7uF and R:51K on SS pin, build a RC delay circuit, it will disable the SPI port for about 200ms when power on, the 10K on CLK ground noise if SCL floating.

The 2 pull resistors need over 10K on 5 pin header module, if they are too low, there is a problem when transfer data from slave to master.



**NOTE:** SPI mode need a special hand shaking mechanism to let your controller know when the requested data ready to clock out, the flow chart show this mechanism below.

SPI transceiver data flow chart:

## Command summery

There are about 53 different commands usable in Digole’s serial display modules, that include:

- 1) Display characters
- 2) Draw graphics
- 3) Drawing decoration
- 4) Fonts
- 5) Command set
- 6) Use EEPROM
- 7) Use flash memory
- 8) Power management
- 9) Operating touch screen panel
- 10) Setting and configuration
- 11) Others

## 256 color code

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F	90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF
E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF

## Display characters

### Display a string (TT.....\x00)

Command: **TT**. following with characters until value 0 received, the value 0 also is the terminator of a regular string. This command display a giving string on the current position, the position adjusted automatically, if the position reach the most right of screen, it move to the beginning of next line of character, the module can calculate the next character line according the current using font’s size.

The value 10 and 13 (\n and \r in C, LF and CR in ASCII table) can move the current position, value 10 move to next line, value 13 move to the beginning of current line, use 10 and 13 move to the beginning of next line.

eg.:

“TTHello\nDigole” output on screen:

```
Hello
    Digole
```

“TTHello\n\rDigole” output on screen:

```
Hello
Digole
```

Arduino lib function: `drawStr(x,y,string),print(char),print(string),print(int).`

### Move current position(TPxy)

Command: **TP**. follow by x,y position, the x,y value is not refer to pixels, they are the column and row value that MCU calculated based on the font’s size (usuall use space size to calculate current font’s size).

The top-left position is: 0,0.

Arduino lib function: `setPrintPos(x,y,0); drawStr(x,y,string);`

### Enhanced move current position(ETPxy)

Command: **ETP**. follow by x,y position in pixels, this function can adjust the text position as pixels on screen. The top-left position is: 0,0.

Arduino lib function: `setPrintPos(x,y,0);`

### Move position to last(ETB)

Command: **ETB**. after each character printed on screen, the current position is adjust to new value, the MCU also remembered the last character's position, if you want print few characters at same position, you can use this function. eg.: print a bold **K**, the command sequence is: "TTK\x00ETB ETO\x02\x00TTK", in here we use move position offset command, it move the last position 2 pixels right.

Arduino lib function: `setTextPosBack();`

### Move position offset(ETOxy)

Command: **ETO**, follow by x,y value in pixels, the range of x,y value is -127~127, it adjust the current position with the relative value.

eg.: if current position is 46, 30, and the x=-10, y=5. after run this command, the new position is:36,35.

Arduino lib function: `setTextPosOffset(x,y);`

### TextCursor(CSd)

Command: **CS**, follow by a value 0 or 1, if d=1, the module will show a small cursor at the current position, otherwise, no cursor displayed.

Sorry, the cursor on function not always work properly for some reason, for example, when the cursor show up, you clear the area where cursor in, the cursor is disappeared, but the module still think it's show up, this will make it massed when module blinking the cursor.

Arduino lib function: `enableCursor(); disableCursor();`

### Text Alignment(ALIGNd)

Command: **ALIGN**, follow a value 0,1 or 2, indicate how to align the text based on the current position in the on coming display string (TT) command, value 0=left alignment (default), 1=middle alignment, 2=right alignment.

## Draw graphics

CGP-----Current Graphic Position

### Set current graphic position(GPxy)

Command: **GP**, follow by x,y value in pixels, this function is same with “ETP” since firmware V3.0 and later, but, in older version before V3.0, the CGP and text position are separated.

Arduino lib function: `setPrintPos(x,y,0);`

### Draw a pixel(DPxy)

Command: **DP**, follow by x,y value in pixels. This function draw a pixel at (x,y) position using foreground color (set by commands “SC” or “ESC”), the pixel also logic operate(draw mode, set by “DM” command) with existing pixel at same position.

This function doesn't change CGP.

Arduino lib function: `drawPixel(x,y);`

### Draw line(LNx1y1x2y2)

Command: **LN**, follow by two coordinates which indicate where the line drawing from and to (x1,y1)(x2,y2) in pixels, the foreground color and draw mode affect this function, also affected by line pattern (set by command “SLP”).

The CGP also move to (x2,y2) after function executed.

eg.: draw a line from (30,40) to (200,300), the command sequence is this: “LN\x1E\x28\xC8\xFF\x2D”, because the value of 300 exceed one byte, you need use 2 bytes format.

Arduino lib function: `drawLine(x1,y1,x2,y2); drawHLine(x,y,w); drawVLine(x,y,h);`

### Draw line to(LNxy)

Command: **LT**, follow by the destination coordinate (x,y) in pixels, this function draw a line from current position to (x,y), everything else same as “draw line” function.

The CGP move to (x,y).

Arduino lib function: `drawLineTo(x,y);`

### Draw rectangle(DRx1y1x2y2)

Command: **DR**, follow by the top-left coordinate (x,y), and the right-bottom coordinate, all in pixels, this function affect by foreground color, draw mode and line pattern.

The CGP move to (x2,y2).

Arduino lib function: `drawFrame(x,y,w,h); //Arduino lib use width and height follow (x,y)`

### Draw filled rectangle(FRx1y1x2y2)

Command: **FR**, this function similar with “draw rectangle”, but will use current foreground color, draw mode and line pattern to fill this rectangle.

The CGP move to (x2,y2).

Arduino lib function: `drawBox(x,y,w,h); //Arduino lib use width and height follow (x,y)`

## Draw circle(CCxyrf)

Command: **CC**, follow by the coordinate of circle center(x,y), then the radius, the “f” is indicate the circle is filled or not, if f=1, the circle is filled, this function affect by foreground color and draw mode, but **not affected** by line pattern. The CGP move to (x,y).

Arduino lib function: `drawCircle(x,y,r,f); drawDisc(x,y,r);`

## Draw image

There are 4 different functions for drawing images, the difference are on the color depth which function can draw:

### 1) Draw black/white image (DIMxywh...d...):

Command: **DIM**, follow by the top-left coordinate (x,y) of the image, then image width (w), height (h), then follow the image data (...d...), each bit represent a pixel in the image.

Data in one byte can't cross different lines, that means, if the width of image is 12 pixels, you need 2 bytes for each line. MSB is on the left side.

This function affected by foreground color and draw mode, that means you can display different color of image use same command, but just set different foreground color before drawing.

CGP: not change.

Arduino lib function: `drawBitmap(x,y,w,h,*data);`

### 2) Draw 256 color of image(EDIM1xywh...d...):

Command: **EDIM1**, everything similar with “DIM”, except following:

- a) one byte represent a pixel, so, the color depth is 256, the color format is (MSB-LSB):**RRRGGGBB** (332)
- b) draw mode and foreground color will not affect it.

CGP: not change

Arduino lib function:`drawBitmap256(x,y,w,h,*data);`

### 2) Draw 64K color of image(EDIM2xywh...d...):

Command: **EDIM2**, everything similar with “EDIM1”, except this function use 2 bytes to represent one pixel, the color depth is 64K, the color format is: (MSB-LSB):**RRRRRGGG** , **GGGBBBBB**(565).

Arduino lib function: no

### 2) Draw 262K color of image(EDIM3xywh...d...):

Command: **EDIM3**, everything similar with “EDIM1”, except this function use 3 bytes to represent one pixel, but only the 6 MSB used for a color, the color depth is 262K, the color format is: (MSB-LSB):00**RRRRRR** , 00**GGGGGG** , 00**BBBBBB**.

CGP: not change

Arduino lib function: `drawBitmap262K(x,y,w,h,*data);`

## Move area on the screen(MAxywhOxOy)

Command: **MA**, follow by the top-left coordinate of the area, then the area width and height, all as pixels the Ox, and Oy are the offset refers to (x,y), it will move the area(x,y)-(x+width,y+height) to new top-left position of(x+Ox,y+Oy), the value of Ox, Oy are -127~127.

This function is useful to scroll screen in 4 directions.

Arduino lib function:`moveArea(x,y,w,h,Ox,Oy);`

### Video Box (VIDEOxywh\x00/\x01....data)

This command let user to send raw image data to the LCD panel directly, after command “VIDEO”, follow by 2 integer data x, y to indicate the top-left position counted as pixels where of the video box, the available value of x,y are from 0 to 65535 but not exceed the LCD panel size, then 2 bytes of value to indicate the box width and height, the available value are from 0 to 255.

After defined the video box, the next byte of value indicate the color depth of each pixel, if value is 0, the color depth is 16BIT(2 bytes data) : **RRRRRGGG GGGBBBBB**, if value is 1, the color depth is 18BIT (3 bytes data): **RRRRRR00, GGGGGG00, BBBBBB00**, except 2.6” IPS module, which only accept 18BIT (3 bytes data) with the format as: **0RRRRRR0, 0GGGGGG0, 0BBBBBB0**.

Then all the coming data from serial port will be sent to the video box directly.

How to exit the video mode? if user want to exit video mode and display other things on the screen, just stop send the data to the display, wait about 120ms, the module will exit video mode, and all following datas are treated as regular commands.

The maximum speed: UART mode-460800bps, I2C->400K bps, SPI-10MHz.

Only setRotation command-“SD” affect the video command, other decoration commands are ignored.

Here is the example code for reference:

```

writeStr("VIDEO"); // "VIDEO" command
writeByte(0); //the X=0 position of top-left for Video window
writeByte(0);
writeByte(0); //the Y=30 position .....
writeByte(30);
writeByte(150); //the width of Video window, here is 150pixels
writeByte(100); //the height of Video window, here is 100pixels
writeByte(0); //the color format is 16bit, if the value is 1, the format is 18bit, doesn't matter on 2.6" IPS display
delay(300); //give the display enough time to init the Video box
for(long int n=0;n<150*100*2*20;n++) //send 20 frames data to display
    writeByte(data[n]);
delay(200); //this delay will cause the Digole display exit the "VIDEO" mode
printText("Thank you!"); //print "Thank you!" on the screen
..... more your code

```

## Drawing Decoration

### Clear screen(CL)

Command: **CL**, clear the screen panel: use current background color to clear the screen.

This function also reset current font to 0, screen rotation to 0, x position to 0, draw mode to 'C', draw window to full screen, line pattern to 0xff.


Arduino lib function: `clearScreen();`


### Set background color(BGC)

Command: **BGC**, specify the one byte value of color (256 color depth,332 format) follow this command. eg.: set a red background: "BGCxE0".

Arduino lib function: `setBgColor(c);`

### Set foreground color(SCc,ESCrgb)

 There are 2 commands to set foreground color: "**SC**" then follow by a byte to set 256 color depth, and "**ESC**" follow by 3 bytes to set 262K color depth, the color format refer to "EDIM1" and "EDIM3" commands.

 Only 2 different value for these display module: 0 and 1.

Arduino lib function: `setColor(c); setTrueColor(r,g,b);`

### Set line pattern(SLPd)

Command: **SLP**, follow by a byte indicate which pixel should display or not, there are 8 bits in a byte, so when drawing line, the module will repeat every 8 bits according to the line pattern value.

eg.: "SLPx55" command will let the draw line/rectangle function to draw a dotted line, because "x55" equal: 0B01010101, if the bit is 0, that pixel will not displayed.

If the line pattern value is: 0B11010111, the drawing line is dashed.

Arduino lib function: `setLinePattern(d);`

### Set draw direction(SDd)

Command: **SD**, then follow by the 0,1,2,3 direction you want, the original direction is 0, direction 1,2,3 represent 90,180 and 270 degree clockwise(also means turn display panel anti-clockwise). if the value out of {0~3}, the final direction is:  $d\%4$ , means, value 4 is 0 direction, and 5 is 1 direction...

Arduino lib function: `setRotation(d); undoRotation(); setRot90(); setRot180(); setRot270();`

### Set draw mode(DMd)

Command: **DM**, follow by a byte of draw mode which only one letter can be used from {C,I,!,~,^,O,o}. Draw mode is used to tell the module how to display the color of pixel using current foreground color operating with the existing pixel, there are 6 modes available:

'C'-Copy, doesn't matter the existing pixel on screen, this mode use current foreground color to over write the pixel, for "TT" command, it also clear the char box as back ground color, all other modes will not clear the char box.

'I'-Or (the "or" letter is not capital i, it's the shift value on key "~"), use current foreground color "OR" with the existing pixel.

'!' or '~'-Not, doesn't matter the current foreground color, it just "NOT" the existing pixel.



'&'-And, use current foreground color "AND" with existing pixel.

'^'-Xor, use current foreground color "XOR" with existing pixel.

and **all other letter**: 'O' or 'o' means "Over write", similar with 'C', but not clear the char box when using "TT". This mode will let you display text on a picture nicely.

eg.: at (20,20), the pixel color is red: 0B11100000, the foreground color is: 0B00011111, if the draw mode is OR, then when you draw a pixel at (20,20), the new pixel color is White, 0B00011111 OR 0B11100000 =0B11111111, this value is White color. But, if you set the draw mode is AND, the new color is Black.

Draw mode will affect all out put on screen except: display image, clear screen and clear draw window.

Arduino lib function: `setMode(d);`

### Set output/draw window(DWWINxywh)

Draw window was embedded since firmware version 3.2 and later, instead of output to full screen, user can set a smaller rectangle area as draw window, then all following output will be showing in this window and the coordinate also refers to the top-left corner of draw window. This ability provide user a new way to relocate an area of content on the screen to different location easily, just change the draw window to the desired location, then done.

Command: **DWWIN**, follow by top-left coordinate value (x,y), then draw window's width (w) and height (h) all value in pixels.

Arduino lib function: `setDrawWindow(x,y,w,h);`

### Reset draw window(RSTDW)

Command: **RSTDW**; remove the current draw window, what the module do is set the new draw window to full screen.

Arduino lib function: `resetDrawWindow();`

### Clear draw window(WINCL)

Command: **WINCL**, clear the draw window use background color.

Arduino lib function: `cleanDrawWindow();`

### Set image's background transparent ("TRANS 0/1")

When we show an image (256, 65K or 262K color) on the screen, the image occupy a rectangle area, this command can change the image shape on the screen, the black pixels in the rectangle area can be transparent, in order to do this, just set the color to be black (00) where want it to be transparent, then set "TRANSx01" command, here is our logo shows on a blue background when TRANS=0 and TRANS=1:



Digole's Logo

TRANS=0

TRANS=1

Note: the drawing modes: "copy/and/or/xor/not" will affect the display image function now in V4.0, that was not affect in earlier version of firmware.

## For Mono display only

### Refresh screen instantly(FS0/1)

Command: **FS**, follow by a byte of value 0 or 1. if value is 0, the module will not refresh the screen until it receive a fresh screen command such as “FS2”, if the value is 1, the module will refresh the screen from internal screen buffer to screen when the module is idle (no more pending commands in receiving buffer) automatically.

This command only available on Black/White display module, the color module always refresh the screen instantly because no screen buffer used in onboard MCU.

If you need update the screen rapidly, disable the auto-refresh will help to avoid the screen flicking: draw all information to the screen buffer in MCU, the refresh the screen at once.

Arduino lib function: `flushScreen(d);`

### Set screen Normal/Inverse(INV0/1)

Command: **INV**, follow a byte of value 0 or 1 to indicate the screen content normal or inverse, this command only available on some monochrome module, and the content is affected instantly.

## Fonts


Only u8glib format font can be used in our module, you can find vary of font data from here: <https://github.com/olikraus/u8glib/tree/master/fntsrc>, the list of available font is here: <https://github.com/olikraus/u8glib/wiki/fontsize>. If none of standard u8glib font meet your requirement, we also provide an online tool to convert image file to custom u8glib compatible font data, the instruction is here: <http://www.digole.com/forum.php?topicID=330>

### Change current font(SFd)

There are 7 fonts pre-installed in onboard MCU, the fonts' code and name are:

Font code	Font name	Font code	Font name
6	u8g_font_4x6	200	User font 1
10	u8g_font_6x10	201	User font 2
18	u8g_font_9x18B	202	User font 3
51	u8g_font_osr18	203	User font 4
120	u8g_font_gdr20		
123	u8g_font_osr35n		
0(default)	u8g_font_unifont		

The module also reserved flash memory space for each user fonts, the available user font space depends on the flash chip installed or not, if none flash chip on module, the user fonts will use MCU flash memory, and memory space for each font is 3584 bytes.

 When you want to change current font, use command: **SF**, follow by the font's code.  
 Arduino lib function: `setFont(d);`

### Download standard user font(SUFnL...d...)

(You can use "SF"+200~203 command to use these fonts which downloaded by this command).

Command: **SUF**, follow by a byte of the index number of user font space, then 2 bytes of data length of font, then the font data.

40ms delay is needed after each 64 bytes of data send out, refer "Write data to flash" command.

Arduino lib function: `downloadUserFont(length, *data, index);` //if your module is: , put: `#define V33` at the top of your sketch

### Download user font to flash chip(FLMWRal...d...)

There is no special command to download font data to flash chip, instead of it, use regular "download data to flash chip" command:**FLMWR** to any address in the flash chip, then use the next command to tell the module use this font. There is no limitation on the numbers and size of font in flash chip, until the memory size of the chip full.

Following **FLMWR**, there **3** bytes to indicate the start address in chip, and **3** bytes indicate the length of data, all MSB first, then follow by all data byte. 3 bytes can access the address/length 0~16,777,215, and the onboard flash chip usually 2MB~16MB.

Note: You need to erase the flash memory space before writing data, please refer to the "erase flash memory command".

40ms delay is needed after each 64 bytes of data send out, refer “Write data to flash” command.

Arduino lib function: `flashWrite(address,length,*data);`

### Use user font in flash chip(SFFa)

Command: **SFF**, follow by 3 bytes of address which the font data in flash chip start from.

Arduino lib function: `setFlashFont(address);`

## Command Set

### What is command set?

The command set is a command sequence which contain one or more commands and data, that do complicated drawing functions on the screen. Save the command set in the flash chip or MCU flash, run this command set when you needed later, using command set will save you lot of hardware and software resources.

**YES:** Only setting and drawing functions (draw Text or graph) can be put in command set. **NO:** Functions to read data from module and write data to EEPROM and flash can't be embedded in command set.

#### CHANGE:

Few functions are little bit different when used in command set than regular:

Commands : **DIM**, **EDIM1**, **EDIM2**, **EDIM3**, these 4 commands will show a picture on screen, in regular usage, the top-left coordinate (X0,Y0) will follow the command, then width, height and image data. But when it in command set, the X0,Y0 value need to use “GP” command to set it prior, the width and height will follow to the 4 commands directly, this change give user the ability to display same images at different location on screen when call child command set from parent command set.

Command: **LT**. Line to command accept destination coordinate in regular usage, but it accept the coordinate offset (value range -127~127) when used in command set.

eg.: “GP\x0A\x10THello user\x00LN\x00\x12\x30\x12LT\xD0\x01LT\x30\x00\xFF”, this command set will display “Hello user” at (10,16) position, then draw line from (0,18) to (48,18) by command “LN\x00\x12\x30\x12” then to (0,19) by command “LT\xD0\x01”(note: 0xD0=-48) then to (48,19) by command “LT\x30\x00”. and end with “\xFF”. the hex value of \xFF is 255 in dec.

### Write command set to flash(FLMWRal...d...)

Use regular write data to flash function to save command font in flash, if flash chip installed onboard, all 2MB memory can be used for command set, the welcome screen is a real command set.

40ms delay is needed after each 64 bytes of data send out, refer “Write data to flash” command.

Arduino lib function: `flashWrite(address,length,*data);`

### Run command set(FLMCSa)

Command: **FLMCS**, follow by 3 bytes of address which indicate the beginning of command set, 3 bytes address format allow the module access all 2MB memory in flash chip.

## Read data from communication port

When you issued commands which need return data, the data can be read from the same communication port.

On UART mode, the returned data use same Baud rate and setting, if the data returned as bulk, the master need to poll the new data on the port continuously or use interrupt when new data received, there is no hand shake signal between master and slave side.

On I2C module, the module will pull the clock line to low when data not ready, then release it, the master will clock the data out then. The hardware I2C port do this handshake automatically, if the master side use software I2C, check the clock line for release first.

On SPI mode, there is no hand shake in general, we use data out line (SDO) on the module (data in (SDI) on the master) as hand shake line, when the data is not ready on module, the module keep the SDO low, and pull the SDO to high when data ready, the master can check this line (SDI on master side), if the line high, pull the SS line to low, and wait at least 10us, then shift 8bit data out. See the chart flow on page 7.

## Use EEPROM

There are 976 bytes of EEPROM memory could be accessed by user in firmware V3.3 and later, erase the EEPROM cell is not needed before writing.

### Write data to EEPROM(WREPal...d...)

Command: **WREP**, follow by 2 bytes of address, 2 bytes of data length (MSB-LSB format), then the data.

Arduino lib function: `writeE2prom(address,length,*data);`

### Read data from EEPROM(RDEPal)

Command: **RDEP**, follow by 2 bytes of address, 2 bytes of data length (MSB-LSB format), after these 8 bytes of command sent to the module, the master controller need to wait the data available on the communication port, read out all desired data from the port.

Arduino lib function: `readE2prom(address,length); read1();`

## Use Flash

### Use flash in MCU

If there is no flash chip installed onboard, you can use the 16KB flash, user can't read out the data saved in the internal flash memory.

### Use flash in flash chip

If the flash chip installed onboard, you can use the full 2MB~16MB flash chip to store welcome screen, user font, command set and user data, all data in flash chip can be read out. the flash in MCU become unusable.

### Write data to flash(FLMWRal...d...)

*This command applicable to internal 16KB flash or external flash chip.*

Command: **FLMWR**, follow by 3 bytes of start address, 3 bytes of data length (MSB, LSB format), then the data. This command can write data to flash chip or internal flash memory.

**Note:** if write data to flash chip, you probably need to erase the desired memory first before writing, but you don't need to erase the memory in internal flash.

A delay is needed after each 64 bytes of data send to module, when writing to internal flash, the module is writing every 64 bytes as bulk, and during the writing time, all new coming data from the communication port is lost. When writing to flash chip, the delay also needed due to the flash chip is a serial device.

According to our test, a 40ms delay on each 64 bytes is working well for both internal and external flash memory writing.

**Also Note:** the module will send value 17 on the communication port when writing to flash chip done, the master controller can poll it and know when the writing done. But, writing to internal flash will not return this value. Here is the sample code in C:

```
void flashWrite(unsigned long int addr, unsigned long int len, const unsigned char *data) {
    unsigned char c, b;
    unsigned long int i;
    write('F'); //write a byte to communication port
    write('L');
    write('M');
    write('W');
    write('R');
    write(addr >> 16);
    write(addr >> 8);
    write(addr);
    write(len >> 16);
    write(len >> 8);
    write(len);
    b = 0;
    for (i = 0; i < len; i++) {
        c = pgm_read_byte_near(data + i);
        write(c);
        if ((++b) == 64) {
            b = 0, delay(40); //delay 40ms
        }
    }
}
#ifdef FLASH_CHIP
    //check write memory done
    while (read1() != 17); //read a byte from communication port
#endif
}
```

Arduino lib function: flashWrite(address, length, \*data);

### Run command set(FLMCSa)

This command applicable to internal 16KB flash or external flash chip.

Command: **FLMCS**, follow by 3 bytes of address which indicate the beginning of command set, 3 bytes address format allow the module access all 2MB memory in flash chip.

### Read data in flash chip(FLMRDa)

This command only applicable to external flash chip.

If the flash chip installed on the board, you can use it to save user data, and read the data when you need it.


Command: **FLMRD**, follow by 3 bytes of address, then 3 bytes of data length, all MSB format.


After this command issued, the master controller can read data from the communication port when data in module ready.

Arduino lib function: flashReadStart(address,length); read1();

## Erase flash memory in flash chip(FLMERa)

This command only applicable to external flash chip.

Only writing data to flash chip need this command, this command can erase only specific range of address on all  color module. Because the erasing on the chip is operating as block, the module will save the useful data in the block to the RAM on screen panel, erase whole block, then restore the useful data back, so, you may see a block of screen at the left-bottom corner show some wild image, that is the data from the erased block.

In  module, there is not enough RAM to save data from flash chip block, so, all data in the block which the address fall into the desired address range will be erased.

Command: **FLMER**, follow by 3 bytes of starting address, then 3 bytes of length which want to be erased. MSB first.

Arduino lib function: `flashErase(address, length);`

## Touch Panel

The touch screen controller onboard is TSC2046, which can control a resistive touch panel. The internal 12 bit A/D, also can be used to monitor a voltage (battery voltage), an analog input (0~2.5V) and the chip temperature.

### Calibrate touch screen(TUCHC)

Even we already calibrated the touch screen before shipping out, you may still need to re-calibrate it after the module installed in chassis. Send command "TUCHC" to module will let it run calibrate function and save the alignment parameters in EEPROM.

Arduino lib function: `calibrateTouchScreen();`

### Read touched coordinate(RPNXYW)

After module received this command, the module will waiting until the touch panel pressed down, and then send the touched position which mapped to screen pixel's coordinate back to master (x,y), always return 4 bytes, 2 integer value, x first then y.

Arduino lib function: `readTouchScreen(); read1();`

### Read a click event(RPNXYC)

Similar with above function, but the module will return the coordinate data after touch panel released.

Arduino lib function: `readClick(); read1();`

### Read touch panel instantly(RPNXYI), check screen pressed

The 2 of above function will drive the module frozen until the touch screen pressed, if you only want to check the touch screen pressed or not, this is the function for the software, it return a pair of out of range value of no press on touch screen.

You also can check a hardware signal on the module when screen pressed, there is a PENIRQ signal on the 9pin header, this signal will go low when screen pressed. This is the easiest way to quote the touch screen if there were a free I/O pin on your master controller.

Arduino lib function: none

### Read voltage(RDBAT)

Connect a voltage on the Vbat pin on the 9pin header, then send command: **RDBAT** to module, the module will return 2 bytes of data of voltage on the Vbat pin, MSB format, the unit is mV, the range is 0~10,000. eg.: if the 2 bytes of value is: 18, 192, the voltage is:  $18 \times 256 + 192 = 4800\text{mV}$ , is 4.8V.

The input impedance is: 10k $\Omega$

Hint: if the measured voltage is over 10V, a 2R voltage divider is needed.

Arduino lib function: `readBattery();`

### Read analog(RDAUX)

Connect the analog to the AUX pin on the 9pin header, then use this command to read it, we didn't adjust the 2 bytes result here, the data range is 0~4095, and represent 0~2.5V.

Use this format to calculate the real voltage:  $V = d \times 2.5 / 4096$ . (d is the reading data)

Arduino lib function: `readAux();`



### Read temperature(RDTMP)

This command read the temperature of the chip, the format to calculate the temperature is:


$T=(653-(d*2500/4096))/2.1$  °C, d is the reading data.

Note, the temperature on the chip may be affected by the backlight heat of LCD screen.

Arduino lib function: `readTemperature();`

## Power management

### Backlight brightness(BLd)

The backlight brightness on all color LCD and MonoChrome GLCD modules  can be adjusted continuously by use command: BL, follow by a byte of value 0~100, 0 will turn backlight full off, and 100 will turn backlight full on.

The backlight on all OLED modules are not adjust-able.

Arduino lib function: `backLightBrightness(d); backLightOn(); backLightOff();`

### Turn screen on/off(SOod)

Command: **SOO**, follow by a byte value 0/1, when d=0, the screen and the backlight will be turned off immediately, that will save much power on the module, this function work on all module.

On most of modules, the module only consume few mA after screen turned off.

The content on the screen unchanged if screen turn off then turn on later.

Arduino lib function: `screenOnOff(d);`

### Turn MCU off(DNMCU)

Even the MCU will enter sleep mode when no pending commands in the receiver buffer, you may also want to turn the MCU into deep sleep mode manually.

Command: **DNMCU**, no following data needed, the module will check if there were more pending commands in buffer before entering sleep, if there were, the module will not enter sleep mode.

The module wake up automatically when new data received, but if the COM mode is I2C, some dummy data are needed to act as waking signal, so, use few `write(0)` then a delay 10ms is a good practice to wake up the MCU from deep sleep.

The screen will keep on, and all content on the screen unchanged when MCU off.

Arduino lib function: `cpuOff();`

### Turn module off(DNALL)


This command put all power off: backlight off, screen off, MCU enter deep sleep, the module will only consume <0.05mA of current, the wake up sequence is same with wake up MCU, the module will restore backlight and put screen on also after wake up, the content on the screen unchanged.


Arduino lib function: `moduleOff();`

## Setting and configuration

### Start screen(welcome screen or splash screen)

Start screen is the first showing when power on, it is used to display your logo or information of the device, the start screen can be modified and disabled by user.

 On the firmware V3.0 and earlier, the start screen only the bitmap on all monochrome modules, if the screen is 128\*64 pixels, the start screen only use 128\*64/8=1024 bytes of in MCU flash memory.

But in the later, we developed color graphic OLED and LCD modules , the in MCU flash memory were not able to store whole color of start screen, at that time, we use command set to show start screen.

You can use our online tool to convert your picture to bitmap start screen here: [http://www.digole.com/tools/PicturetoC\\_Hex\\_converter.php](http://www.digole.com/tools/PicturetoC_Hex_converter.php).

### Enable/disable start screen(DSSd)

Command: **DSS**, if the following value is 0, the start screen is not show up on next power on.

Arduino lib function: `displayStartScreen(d);`

### Configuration show on/off(DCd)

In default, the module will show start screen when power on, and also show the current COM mode after start screen showed up, that will tell you what is the Baud on UART mode or the slave address on I2C mode.

If you want to manage this configuration show on the screen, use command: **DC**, then follow by a byte value 0 or 1, if d=0, the configuration will not show on the screen on next power on.

Arduino lib function: `displayConfig(d);`

### Download start screen to module(SSSI...d...)

Command: **SSS**, follow by 2 bytes of data length of start screen, then the data, as described before, the data structure are different for monochrome module and color module.

In V3.2 and earlier version on color module, the command set also need 2 bytes of data to indicate the command set length, when you downloading this format of start screen to module, 2 bytes of length follow to **SSS** to indicate the length of rest data, and in the rest of data, the first 2 bytes to indicate the length of command set, their relationship is: SSS (length+2) (length) (...data...).

Arduino lib function: `downloadStartScreen(length,*data);`

### Change I2C address(SI2CAa)

When you connect multiple modules on a I2C bus, every slave modules MUST be assigned with different address, this function can change the default address of 0x27 to other value.

This command only work at I2C COM mode, you can't use it at UART or SPI mode.

Command: **SI2CA**, follow by a byte of new address. The module use the new address instantly, it also save this new address in internal memory, you don't need to change it on the next power recycle.

Arduino lib function: `setI2CAddress(a);`

### Set SPI mode(SPIMD0~3)

There are 4 mode for SPI, based on the Clock polarity and phase, the default is mode 0 (except V3.3, was mode 2). Command: **SPIMD**, follow a byte to indicate the new SPI mode, the module will use the new mode on next power up. This command only available on firmware **V3.4 and later**.

### Change UART baud(SBrate)

The module always on 9600 8N1 baud rate when power on if UART selected, and can be changed by using command: **SB**, the follow vary bytes of baud rate, the module accept 2 format of data:

- 1) unsigned long int: 4 fixed bytes, MSB first.
- 2) numbers: vary length.

eg.: set the baud rate to 115200: "SB\x00\x01\xC2\x00"—long int format, or "SB115200"—number format.

The module will not save the new baud rate in memory, so, you always need to start your mater circuit on 9600 rate first, then send the change baud rate command, then adjust the master's rate to new.

Arduino lib function: `DigoleSerialDisp mydisp(&Serial, rate);`

### Config universal character LCD adapter(STCrcr\x80\xC0\x94\xD4)

Our universal character LCD adapter can work with different size of character LCDs: 0801,0802,0804, 1601, 1602, 1604, 2001, 2002, 2004, 4001 and 4002 if the LCD controller is KS0066U/F / HD44780 or compatible chip.

The default set is 1602, if your LCD is other than 1602, you need to use this command: **STCR**, follow by a byte for column and a byte for row, the rest of 4 bytes is fixed for KS0066U/F / HD44780 LCD controller.

eg.: for 2004 LCD: "STCR\x14\x04\x80\xC0\x94\xD4".

Arduino lib function: `setLCDColRow(c,r);`

### Config universal graphic LCD adapter(SLCDx...)

In the new version of Universal GLCD adapter V2, the firmware can recognize the KS0108, ST7920 and ST7565 (SPI) by itself. It also support ST7565 in parallel mode by using command of "SLCD3" and following 8 bytes of init parameters, that pinout are same with for ST7920.

Because the ST7565 chip used widely in 12864 LCD panels, and it's very flexible of configuration, you need set at least 8 bytes of init parameters, if the ST7565 (SPI) not working well with this adapter, please use command "SLCD2" following by 8 bytes of init parameters also:

#	valid value(HEX)	Function	Detail description
1	A6 / A7	Control the pixel normal or reverse	A6=Normal(default), A7=Reverse
2	A2 / A3	Set LCD bias	A2=1/9 bias (default), A3=1/7 bias
3	A0 / A1	Set the horizontal scan direction	A0=Normal(default), A1=Reverse
4	C0 /C8	Set the vertical scan direction	C0=Normal(default), C8=Reverse
5	20~27	Select internal resistor ratio(Rb/Ra)	25 (default)
6,7	81+[0~3F]	Set screen contrast	81+24 (default)
8	40~7F	Set RAM start address, 40 start 0, 7F start 3F	40 (default)

The default init parameter set is equal to this command string: "SLCD2\xA6\xA2\xA0\xC0\x25\x81\x24\x40".

### Adjust LCD contrast(CTx)

Command: **CT**, follow by a byte of value 0~100, this command only effective for 128\*64 GLCD with ST7565 controller, The contrast on GLCD use KS0108 and ST7920 controller only be adjustable by a hardware pot.

Arduino lib function: `setContrast(x);`

## Others

### Delay a period(DLYx)

This command only available since V3.9, but this is a bug in V3.9: it will be halt if on I2C/SPI mode, fixed in V4.0.

Command: "**DLY**", following with a byte of delay period, value 1 for about 0.25s.