MGL(S) - 240128
DISPLAYS

APPLICATION NOTE

Prepared for

TRIDENT MICROSYSTEMS LTD
PERRYWOOD BUSINESS PARK
HONEYCROCK LANE
SALFORDS
REDHILL
SURREY
RH1 5JQ

By

AND SOFTWARE LTD
4 FOREST DRIVE
THEYDON BOIS
ESSEX
CM16 7EY

CONTENTS

1. INTRODUCTION

1.1 BACKGROUND

This application note is to help designers incorporate the MGL(S)-240128 display module into their products.

The intention is to supplement rather than replace the existing data sheets for the display module and its Toshiba controller. It concentrates on the areas where this existing documentation was found to be weakest. Although the Toshiba controller data sheets is the main source of technical detail it is inevitably a general description covering all the controller's potential uses and is not specifically targeted for its use in the module in question. It follows that some parts of the data sheet describe the use of the controller in ways which are not possible or appropriate in the finished module. This can make the existing documentation confusing and hard to follow and this application note tries to clear up some of the uncertainty in this area.

The information is presented in an informal style with examples, rather than as a fully rigorous treatment.

1.2 EXAMPLE DESIGN

The information presented is based on the experience of one design team when including the module into a new product, a scientific instrument. In this application the module was used as the main display element, and needed to show:

- A full screen bit-mapped graphics logo at power up.
- A variety of menu options, each shown as text framed in a rectangular box.
- A variety of tabulated results in text but with some items highlighted by being displayed in inverse.
- Titles in a larger font size than the standard.
- Results shown with subscripts (i.e. smaller characters displaced downwards by part line spacing).
- Graphs of results with annotated axes and large font titles.

Where relevant, for illustration, a brief summary of the options and parameter settings used in this example design are given in each section.

1.3 DRIVING THE DISPLAY FROM A PC

The appendices include details of how to use a low cost proprietary parallel I/O card (Code AM11M from MPS Ltd) to drive the module from a PC. For simplicity this is based on using the card's parallel I/O lines to control and drive the data bus rather than connecting it directly to the PC bus. Sample 'C' interfacing routines for driving the card to generate some simple displays are also given.

1.4 GLOSSARY OF TERMS AND ABBREVIATIONS

LSB	Least Significant Bit or Least Significant Byte
MCU	Microcontroller
MSB	Most Significant Bit or Most Significant Byte
RAM	Random Access Memory (i.e. Read/Write Memory)
ROM	Read Only Memory
VDD	The nominal 5 volt supply fed to the module
VO	The Operating Voltage used to derive the signals which drive the display glass. A relatively large negative voltage (-15->-20 volts) is usually required. The details depend on the fluid used in the module and on temperature.
VSS	The nominal 0 volt supply fed to the module

1.5 REFERENCES

The following documents give background information on the module and the Toshiba controller on which it is based.

Reference 1

Document	T6963C DATA SHEET
Doc. No.	-
Date	-
Version	-
Author	-

Reference 2

Document	VARITRONIX - Liquid Crystal Modules (data sheet / short form catalogue)
Doc. No.	-
Date	11/96
Version	-
Author	Varitronix

2. AN OVERVIEW OF THE MODULE

2.1 PHYSICAL COMPONENTS

The module is a single PCB, with the LCD glass mounted on the front surface and a number of components mounted on the rear. These are:

- The Toshiba T6963C controller
- 2* T6A40 Toshiba drivers
- 3* T6A39 Toshiba drivers
- An 8 kbyte RAM
- A number of sundry components including a main crystal (approx. 6.0 MHz) for the T6963C controller.

Connections to the module are made through a main 18 way 0.1 inch pin header.

In addition, for backlit modules, there is an additional 2 way 0.3 inch pitch pin header which is used to carry power for the backlight.

2.2 ELECTRICAL OVERVIEW

The drivers (T6A40 and T6A39) are used to extend the controller's I/O to allow it to address the necessary 240 columns and 128 rows.

The RAM is used by the T6963C and no connections to it are available via the external connector.

The RAM is used to store the graphics, text and font data which decide the pixels that appear on the display.

Commands are available by which the user can fix how much of the total 8 kbytes of available RAM space is allocated to each of these individual memory areas. More details in later sections.

2.3 SETTINGS FIXED BY HARDWARE PINNING IN THE MODULE

Hard-wiring on the module PCB is used to configure the controller to suit the LCD glass used. The details are given below.

PIN NAME	SETTING	MEANING
DUAL	H	1 LCD Screen (makes sense - there is only one LCD glass on the module)
MDS MD0 MD1	H L L	16 Lines i.e. 128 vertical dots (each line is 8 dots high)
MD2 MD3	L H	40 columns i.e. 240 horizontal dots (each column is assumed to be 6 dots wide, but see discussion in section 5)
FS0 FS1	L H	6 x 8 characters (see note below)

Note that the FS1 input is brought out to pin 18 on the external connector but is also pulled up within the module PCB typically by a 10K resistor to VDD. The default 'logic high' that this generates selects the 6 x 8 font as the default. Grounding this pin on the external connector will bring FS1 low and will therefore switch to an 8 x 8 font setting. See section 5 for a discussion of rows and columns.

2.4 PIN OPTIONS IN THE EXAMPLE DESIGN

It was decided to pull FS low to select an 8 by 8 font. This simplified the graphics line drawing routines (see section 5).

3. HARDWARE INTERFACING

3.1 PIN SIGNAL EXPLANATIONS

External connector pin	External connector signal name	T6963C signal name	Purpose / Remarks
1	FG	--	Frame Ground. This is actually the metallic frame around the LCD glass. It is not internally connected to VSS. It is also not related at all to the T6963C 'frame' signal FR where 'frame' refers to a period of the LCD drive waverform. In general it makes sense to connect FG directly to VSS, which is conveniently the next pin on the connector.
2	VSS	VSS	0 volt ground for the VDD power supply
3	VDD	VDD	The nominal 5 volt feed to the module. Needs to be held within the range 4.5 to 5.5 volts.
4	VO	--	Operating Voltage. This is used to derive the voltage which drive the LCD glass. A negative supply is used to energise the glass is derived from the difference between VO and VDD. The optimum value of VO depends on the type of fluid in the LCD and is significantly temperature dependent. In most designs the voltage fed to VO will therefore need to be variable to allow users to set the best contrast / viewing angle at any actual operating temperature.
5	/WR	/WR	Write enable. This is a conventional low going write signal for the data transfer bus.
6	/RD	/RD	Read enable. This is a conventional low going read signal for the data transfer bus.
7	/CE	/CE	Chip enable. This is a conventional low going chip enable signal for the data transfer bus.
8	C/D*	C/D*	Control / Data* selection signal. This selects whether accesses across the data bus are command / status accesses (with C/D*=HIGH) or data accesses (with C/D*=LOW).
9	/RST	/RST	Reset. This is a conventional low going reset signal. It will need to be driven low for at least 5 oscillation clock periods (i.e. at least 1.2 microseconds) after the power supplies have established.
10->17	DB0->DB7	DB0->DB7	The data bus. Conventional bi-directional bus, controlled by /CE, /WR, /RD and C/D* signals.

18	FS	FS1	Font Select. Note this is FS1. FS0 is not brought out to the external connector but is fixed low by wiring in the module. The FS1 signal is pulled up within the module PCB typically by a 10K resistor to VDD. The default 'logic high' that this generates selects the 6 x 8 font as the default. Grounding this pin on the external connector will bring FS1 low and will therefore switch to an 8 x 8 font setting. See section 5 for a discussion of rows, columns and fonts.

3.2 TYPICAL BUS SYSTEMS

The data bus used to send commands and data is conventional.

The basic method of connecting the module into the address space of an MCU is illustrated as follows:

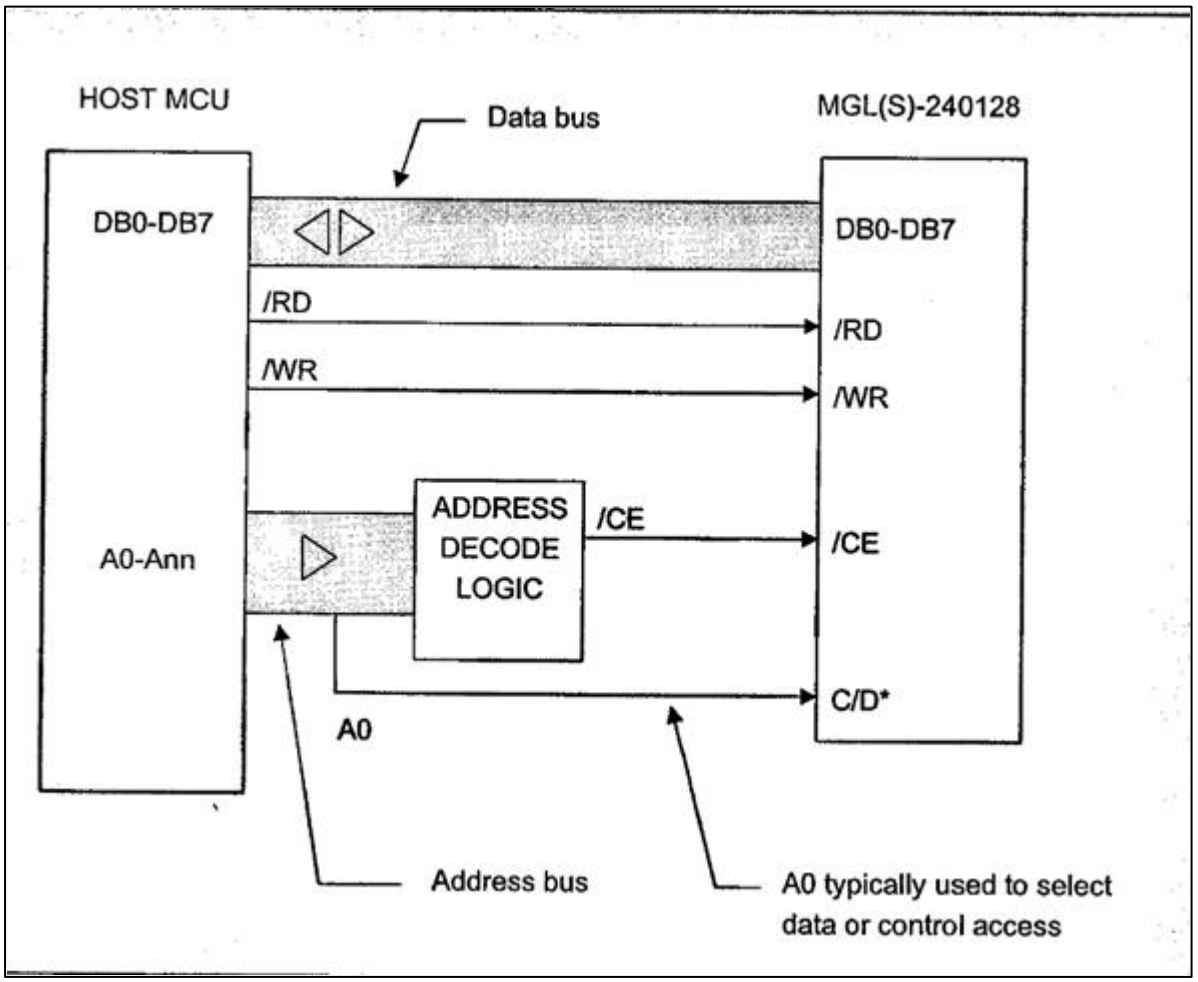


Figure 1: Typical hardware interfacing scheme

3.3 THE EXAMPLE DESIGN

The interfacing circuitry used for connecting the display module to the host microcontroller which was an Hitachi H8/300H device.

3.4 WAIT STATES

If a fast processor is used as the host MCU it may be that the processor wait states will need to be used to give the display time to pick data from or put data onto the bus.

In practice the need for wait states will typically occur when the overall processor's write cycle takes of the order of 150 nanoseconds. For details see the detailed timing in ref 1.

3.4.1 HARDWARE INTERFACING AND WAIT STATE HANDLING IN THE EXAMPLE DESIGN

The host MCU in the example design was a member of the Hitachi H8/300H family (the HD6413003RF device). The H8/300H includes on-chip address decoding logic and this was used to generate the necessary chip select signal to the module. The HD6413003RF was running at an oscillation frequency of 32 MHz (16 MHz internal) and consequently the normal read and write bus cycles would have been significantly too fast for the module to respond correctly. The HD6413003RF also includes an internal wait state controller and this was used to add a single wait state in each access to the module, so that the overall cycle times (from chip select drop to chip select raise) were then 250 nanoseconds for both read and write cycles.

3.5 OPERATING VOLTAGE

No pixels will appear on the display unless a suitable negative operating voltage is applied to the VO pin (pin 4 on the module header).

This is in contrast to many smaller display modules such as those based on the Hitachi 44780 or compatible controllers, where the 5 volt supply is often sufficient to drive the LCD glass and no additional drive voltage is required. The large number of pixels of the MGL(S) implies a very high multiplex ratio (1:128), so that a higher drive voltage is needed to produce sufficient RMS voltage to activate the pixels.

The optimum voltage to be used varies with the type of fluid used in the display and with temperature. The temperature variations are quite marked so it is common practice to provide a potentiometer fed supply for VO so that users can themselves set the drive level to suit their preference in any operating situation. For guidance on VO ranges for types of fluid, see ref 2 which shows recommended voltages for the various differing members of the MGL(S)-240128 family.

The quoted voltages are given with respect to VSS, although the effective drive voltage to the LCD glass is actually VDD-VO.

3.5.1 OPERATING VOLTAGE IN THE EXAMPLE DESIGN

The design example used a nominal supply voltage of -16.5 volts.

3.6 RECOMMENDED POWER UP SEQUENCE

It is necessary to apply the VDD supply to the module significantly before the VO supply. If this sequence is not followed a phenomenon known as 'reverse domain twist' can occur and the result can be that pixels come on when they shouldn't. Typically when this does occur, the effect is almost unnoticeable immediately after a power on when only a few such pixels may be seen but the effect gradually worsens over a few minutes. Close examination shows that an affected pixel initially turns on in part (such as the top left hand corner of the pixel) only, but over the course of a few minutes the effect spreads across the whole pixel, and is unaffected by the pixel being rewritten or refreshed by the controlling software application.

3.6.1 POWER UP SEQUENCING IN THE EXAMPLE DESIGN

In the example design the VO power supply was simply slugged to produce approximately 100 milliseconds delay after the application of the VDD supply.

4. MODES

RAM contents can be read and written by the application using appropriate commands and indeed writing data to the RAM is the fundamental way that applications get their data onto the LCD screen.

The T6963C is a little unusual in that it typically uses differing areas in the same RAM space for its text, character generator look-up tables and low level graphic pixel data.

The details of how the RAM is interpreted by the T6963C to determine which pixels are displayed depends on the 'mode'.

4.1 WHAT THE DATA SHEET MEANS BY 'MODE'

There is some confusion in the T6963C data sheet's use of the terms 'mode' and/or 'display mode'.

A clear view can be obtained by focusing on the two distinct commands involved. These are shown in the following table along with some new suggested names in addition to the corresponding T6963C data sheet names:

Command word (hex)	Suggested name	T6963C name	Purpose / Remarks
80->8f	Pixel combination method	MODE SET	All commands in this range force a particular way of combining graphics and text.
90->9f	Display enables	DISPLAY MODE	All commands in this range force one particular combination of 'enables'. These cover the display as a whole on or off, the cursor and its type, and separate overall enables for text and graphics.

4.2 PIXEL COMBINATION METHODS

There are four basic ways of combining graphics, font and text data. Of these, three treat the graphics data as raw pixel data and mix this with the text pixels. The remaining method is called 'TEXT ATTRIBUTE' mode and here the data in the graphics area is not treated as raw pixels at all but instead sets attributes which change the appearance of the text from the text area. These four main methods are described in this table:

Mode	Comments
'OR' mode	The actual composite screen pixel pattern is built by logically *'OR'ing* the graphics and text related pixels together. This is illustrated in Figure 2 below and is what might typically be used to display a graph for example, with the graphics pixels used to form the axes and the plotted curve, but the text area being used for annotating the plot with title, scales and so on.
'EXOR' mode	Here the screen pixels are determined by an *'EXCLUSIVE OR'* function on the graphics-related and text-related pixels. This might be used to show characters in inverse i.e. white on black, by setting to '1' all the graphics pixels in the area corresponding to the text.
'AND' mode	Here the screen pixels are determined by an '*AND*' function on the graphics-related and text-related pixels.
'TEXT ATTRIBUTE' mode	This mode differs significantly from the others in that the data in the graphics area does not correspond to pixels at all, but is used to set attributes for the text being displayed. The attributes that can be set are various combinations of *inhibit* (i.e. don't display that character), *inverse* (i.e. show it white on black) and *flash*.

4.3 PIXEL COMBINATION EXAMPLE - 'OR' MODE

To illustrate how the pixel combination logic operates, Figure 2 shows what happens in a typical case - so called 'OR' mode. In this case, what is shown in any area of the screen is a logical 'OR' combination of the low level graphics pixel data for that area wit the text pixels for that area. The text pixels are determined by the character generator area contents for the text character in question. This 'OR' mode would be suitable for graphics, maps, tables and other displays of mixed graphics and text.

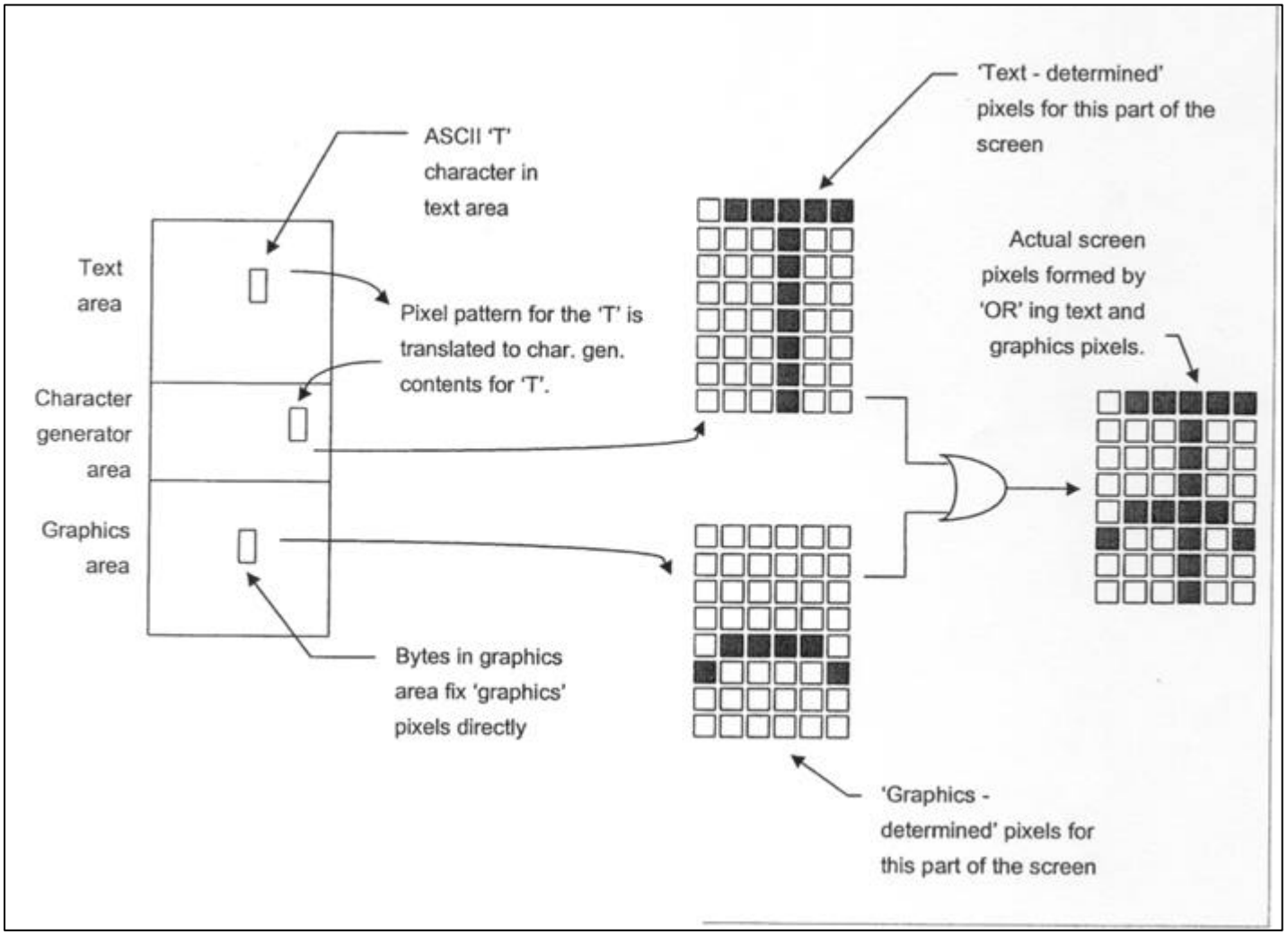


Figure 2: Pixel combination in 'OR' mode

4.4 CHARACTER GENERATOR

The T6963C includes a built in character generator in internal ROM which can be used to generate codes for the lower half of the character map (i.e. for codes from hex 00 to hex 7f). In this character map, the characters from hex 00 to hex 5f are similar to ASCII characters from hex 20 to 7f. A translation for ASCII characters can therefore readily be effected by subtracting hex 20 from the ASCII character values before writing them into the text area. Switching between the ROM character generator and the RAM character generator for the lower half of the character map is done via bit 3 in the pixel combination command (in the range hex 80 to hex 8f).

Note that the top half of the character map is always in RAM.

4.5 LIMITATIONS OF THE AVAILABLE PIXEL COMBINATION METHODS

Although at first glance there appears to be plenty of flexibility for combining text and graphics to build useful practical displays, there are some significant pitfalls. Remember that whichever method is selected at any one time applies to the whole display - it is therefore not possible to set, for example 'TEXT ATTRIBUTE' for one part of a display and 'OR' mode for the remainder. It is however possible to switch between methods at different times, to suit different display formats for example.

The next sections highlight some of the points to bear in mind when choosing a mode for your display format.

4.6 LIMITATIONS OF 'TEXT ATTRIBUTE' MODE

In 'TEXT ATTRIBUTE' mode there is really no effective way to exercise arbitrary control over individual pixels. In some cases this limitation can be overcome by using encoding appropriate graphic elements into any spare unused cells in the character font. An example here would be a set of line and corner elements for displaying boxes around text. Note that this approach does have limitations - each element must for example, be a whole number of character cells in size.

In this mode all the text must be shown in the same font, and differing fonts cannot be mixed in one display format. Again there is a work around here in that unused cells in the character map can be loaded with patterns which can be used to show or build other sized characters. As an example 10 otherwise unused character cells could be used to show the characters '0' to '9' in a smaller font, suitable, say, for subscripts. Note however that this approach is again limited because the smaller characters are still embedded in the full size character space and therefore cannot be displayed closer together than the normal size characters.

4.7 LIMITATIONS OF GRAPHICS BASED MODES

In the three methods which treat the graphics data as true pixel data (i.e. 'AND', 'EXOR' and 'OR' modes), it is not possible to invoke text attributes. Inverse, or flashing text cannot therefore be generated automatically. Some attributes can be implemented by suitable low level actions (e.g. regularly re-writing flashing text alternately as blanks), but this will inevitably have an overhead in the application software and you will need to consider this carefully in your design.

4.8 LIMITATIONS OF ALL CHARACTERS DISPLAYED FROM THE TEXT AREA

Every character displayed using the character generator to translate characters from the text area will be aligned on the standard row and column grid. This can be inconvenient particularly in display formats which try to get a lot of information on the screen at once. When labelling the axes of a graph, for example, the text needs ideally to be placed a pixel or two away from the axis to give a tidy result, but the column and row grid may not allow this.

4.9 CHOOSING A PIXEL COMBINATION METHOD

If your display requires arbitrary single pixels or groups of pixels, you will probably need to use 'OR' mode and implement any special attributes yourself by low level actions in your application.

If you need general text in several font sizes, again you will probably need to use 'OR' mode so that you can have your application write the non-standard size characters by building them up from their individual constituent pixels.

If you need to place text characters anywhere other than on the fixed rows and columns you will also probably need to use 'OR' mode, and write the characters by building them up with pixels.

In summary, it would appear that most practical formats will need to use 'OR' mode and will need to build up from low level pixels, any characters which are non-standard sizes or that need to be positioned anywhere other than on the standard row and column grid.

4.9.1 THE PIXEL COMBINATION METHOD CHOSEN FOR THE EXAMPLE DESIGN

The need to get the maximum information on the display in an attractive form, dictated that various sized fonts where needed and that many characters needed to be placed off-grid. In addition several formats needed to show graphs and many needed boxes around text items. All these considerations meant that 'OR' mode was the only sensible choice.

In fact it was decided that the need to be able to place even some of the standard size characters off-grid, meant that even these would have to be built up by low level pixel writes, so this low level approach was used for every character in every format. The text area and its associated character generator area was therefore never used and text was never enabled.

5. MEMORY OVERVIEW

The module has 8 k bytes of RAM fitted, so valid addresses range from hex 0 to hex 1FFF.

5.1 TEXT AREA

If the 8 by 8 font is selected, then there are 30 characters across the screen and to cover the whole display the text area size must therefore be:

30 * 16 = 480 bytes (hex 01E0)

In 6 by 8 font situations, there are 40 columns so the allocations become:

40 * 16 = 640 bytes (hex 0280)

5.2 GRAPHICS AREA

If the 8 by 8 font is selected, then in the three true graphics modes, each bit controls 8 pixels so each row of pixels requires 30 bytes. To cover the whole display the graphics area size must therefore be:

30 * 128 = 3840 bytes (hex 0F00)

In 6 by 8 font situations, each graphics byte covers only 6 pixels, so 40 bytes are needed to cover each row, and the corresponding allocations for the whole display must be:

40 * 128 = 5120 bytes (hex 1400)

Note that in TEXT ATTRIBUTE mode, each byte defines the attributes for a complete character, just as each byte in the text area maps to a complete character. The graphics area allocations in TEXT ATTRIBUTE mode are therefore identical to the text area allocations and are significantly smaller than in the three true graphics modes.

5.3 FONT AREA

The RAM area needed to define a character in the font map doesn't depend on whether the font selected is the 6 by 8 or the 8 by 8 - unused bits simply become 'don't cares'.

If the internal ROM character generator is used then the effective maximum size of the character generator RAM will be the 128 character codes from hex 80-hex FF:

8 * 128 = 1024 bytes (hex 400)

If the internal character generator is not enabled then the maximum size of the character generator RAM will rise to:

8 * 256 = 2048 bytes (hex 800)

The internal ROM character generator is enabled or disabled by the state of bit 3 in the pixel combination method setting command (in the range hex 80 to hex 8f).

5.4 SUGGESTED MEMORY MAP

The 8 k bytes provided are sufficient to allow the maximum allocations for all these areas and a suggested usage is therefore as follows:

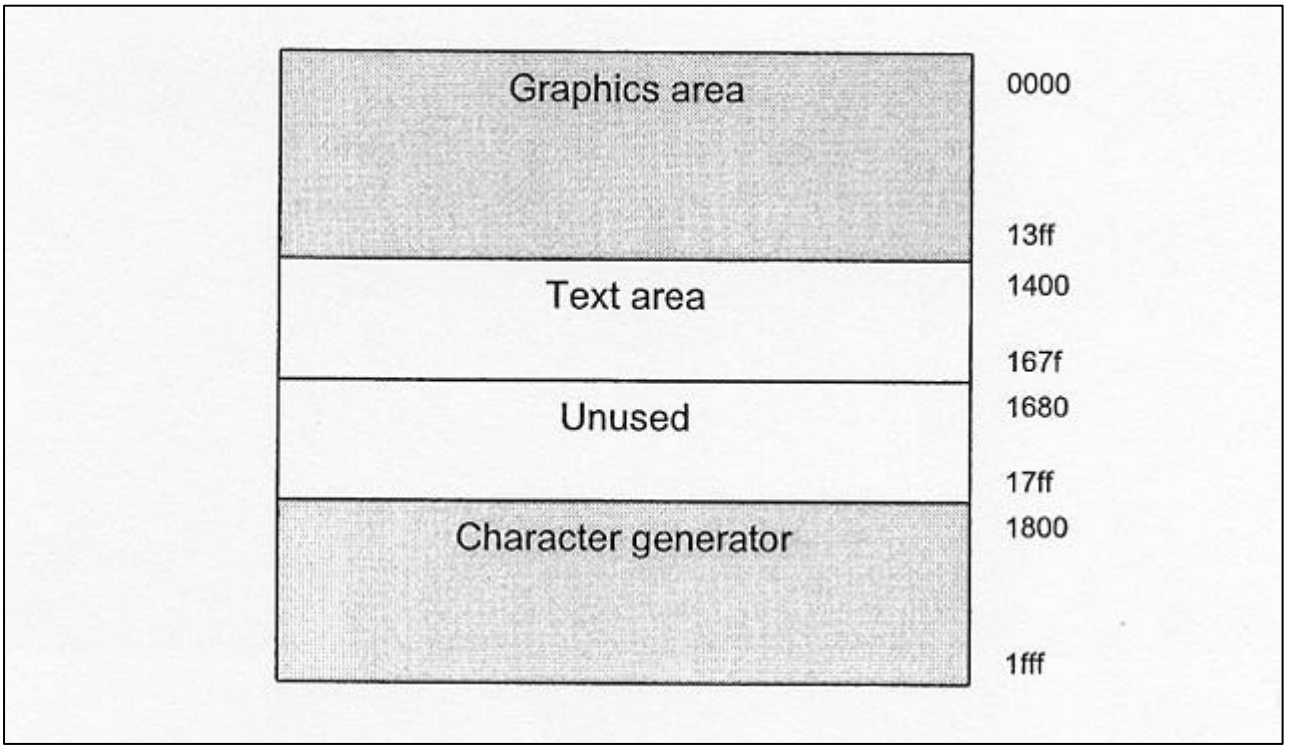


Figure 3: Suggested memory allocations

5.5 RAM IMPLICATIONS OF COLUMN CONFIGURATIONS

The T6963C data sheet refers to three differing ways in which the number of columns can be configured:

- Pin-programmable. Via the MD2 and MD3 pins (fixed in the module to 40 columns)
- Text area columns setting via the text area set command (hex 41)
- Graphics area columns setting via the graphics area set command (hex 43)

The detailed functions of these various settings are discussed in the sections which follows.

5.6 PIN PROGRAMMABLE COLUMNS

The data sheet is not specific about exactly what the hardware programming set by MD2 and MD3 determines.

One might have expected that it would set the number of character cells across the screen, and in this case, it would be expected that switching font widths might alter the effective number of horizontal pixels driven, because for example 40 columns of 8 pixels wide characters corresponds to 320 horizontal pixels, whereas 40 columns of 6 pixels corresponds to 240 pixels. In practice changing the font width setting does not appear to affect the level of drive to each pixel or the number of pixels driven, and this would suggest that the drive signals are unaffected. It would therefore appear that the hardware column settings simply tell the controller how wide the display is, assuming 6 pixel wide columns regardless of the actual selected font setting.

In the case of the MGL(S) 240128 the pin-programmable column setting is fixed at 40 and there are 6 times as many (240) horizontal pixels so this appears to make sense.

Trident may be able to clarify here.

5.7 TEXT AREA COLUMN SETTING

The parameter set by this command tells the T6963C how much to add to any text RAM address to get to the corresponding address for the next row.

It therefore fixes the separation in text RAM between consecutive rows on the LCD. Rather oddly, it doesn't actually fix the length of rows themselves which remain at the full display width, and this can result in characters being repeated on the display in multiple positions.

Taking a relatively extreme example to illustrate this, assume that this 'text area columns parameter' is set to 4 columns.

Writing the character 'a', to the first text address (i.e. the text home) will cause an 'a' to appear at the top left hand corner of the LCD:

Writing 'b', 'c' and 'd' to the next three bytes respectively will add these to the display:

```
abcd
```

This is very much as expected. However if we now write 'e' to the next address, it will be shown on the LCD twice. This is because the text column parameter setting is now '4' and the 'e' is 4 bytes on from the start of the first line, so the 'e' is now regarded as the first character of the next line, and it therefore appears under the 'a'. Because the column setting doesn't actually shorten the length of the rows it will also appear on the first row after the 'd':

```
abcde
e
```

It we continue to write characters to consecutive addresses in this way, 'I' will then appear as the first character on the third row, as well as the 5th character on the second row and the 9th character on the first row:

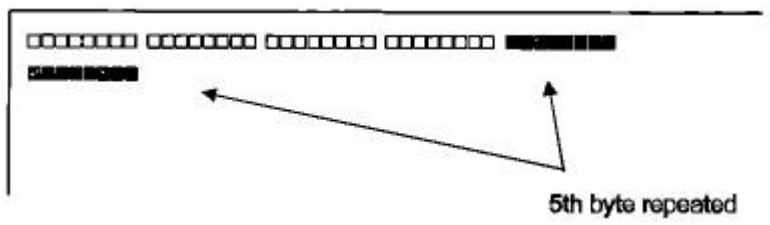
```
abdcefghi
efghi
i
```

Although it is conceivable this slightly curious behaviour might have some value in some applications in the vast majority it doesn't and the only sensible values to set are 40 for 6 dot wide font and 30 for the 8 dot wide font. These settings will give a clear, efficient mapping between text RAM addresses and LCD character cells.

5.8 GRAPHICS AREA COLUMN SETTING

This is very similar to the text column setting parameter described above, but instead of setting the RAM separation of consecutive rows of characters, it sets the RAM separation of consecutive rows of graphics pixels. Like the text columns parameter it also does not fix the length of the rows, so setting a value shorter than 40 (for 6 dot wide fonts) or 30 for 8 dot wide fonts will result in repeated pixels.

Taking a similar example to that above where the graphics column setting is 4, and writing 5 bytes of data starting at the graphics home address, would affect the display as shown. For clarity the fifth byte is chosen to be all ones with all the others all zeroes:



Just as for the text columns, the upshot of all this is again that it only makes sense to select 40 columns for 6 dot wide fonts and 30 columns for 8 dot wide fonts.

5.9 RAM IMPLICATIONS OF FONT SELECTIONS

The FS signal on the external connector allows control of the FS1 input to the T6963C. Pulling this low sets an 8 dot wide font, leaving it high sets a 6 dot wide font.

These settings basically fix the number of horizontal pixels that each byte of the text area or graphics area control. Note that this fixes the display width of graphics bytes as well as text bytes. This means that when an 8 dot wide font is selected each bit in each graphics area byte influences one pixel and when a 6 dot wide font is selected only the lower 6 bits influence the display and the top two bits have no effect.

This is illustrated in the two diagrams which follow. The first shows the situation corresponding to an 8 dot wide font (i.e. FS=LOW) and the second shows how this changes when FS is left pulled HIGH. In these examples it is assumed that the number of graphics columns has been set to 240 divided by the font width, so there is no premature wrapping to the next line.

5.10 GRAPHIC BITS TO PIXEL MAPPING WHEN 8 BY 8 FONT SELECTED

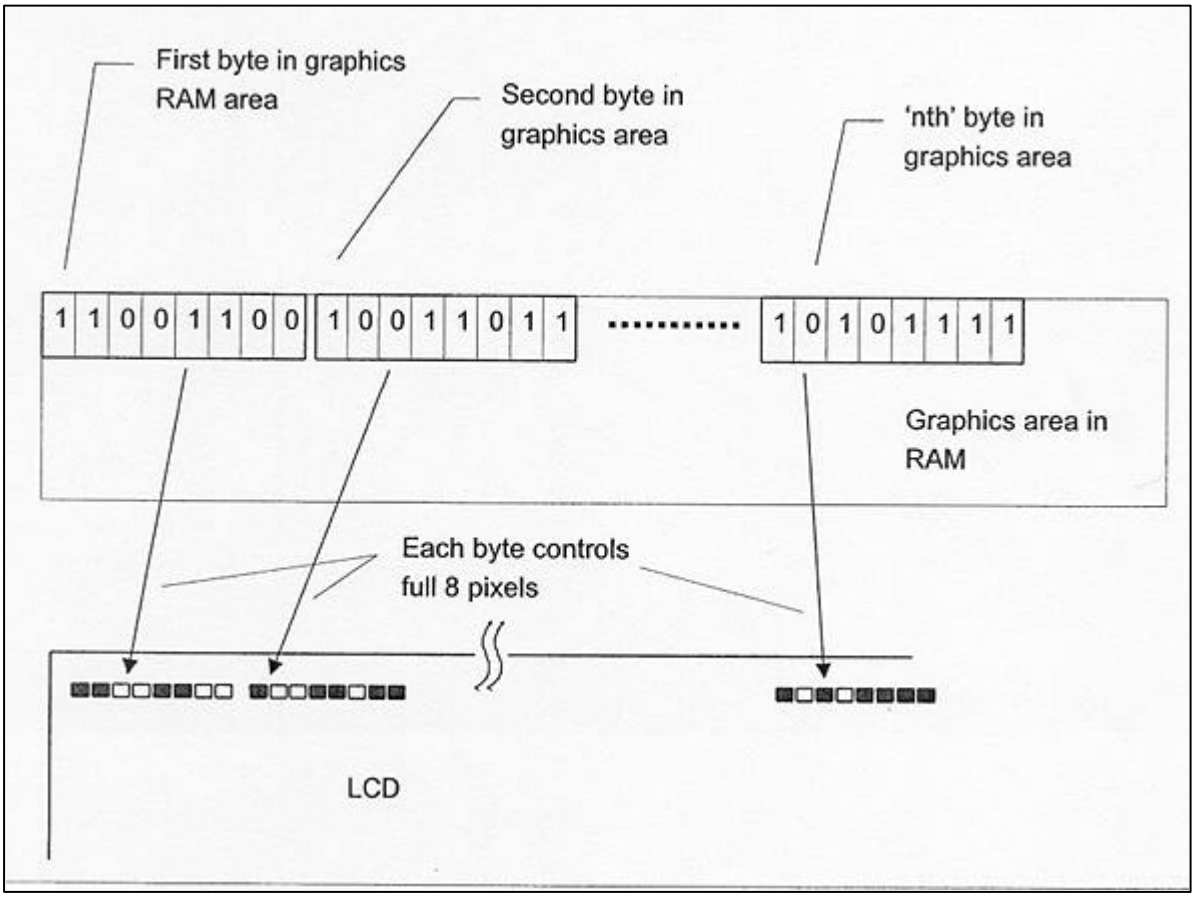


Figure 4: Graphic bits to pixel mapping - 8 by 8 font

5.11 GRAPHIC BITS TO PIXEL MAPPING WHEN 6 BY 8 FONT SELECTED

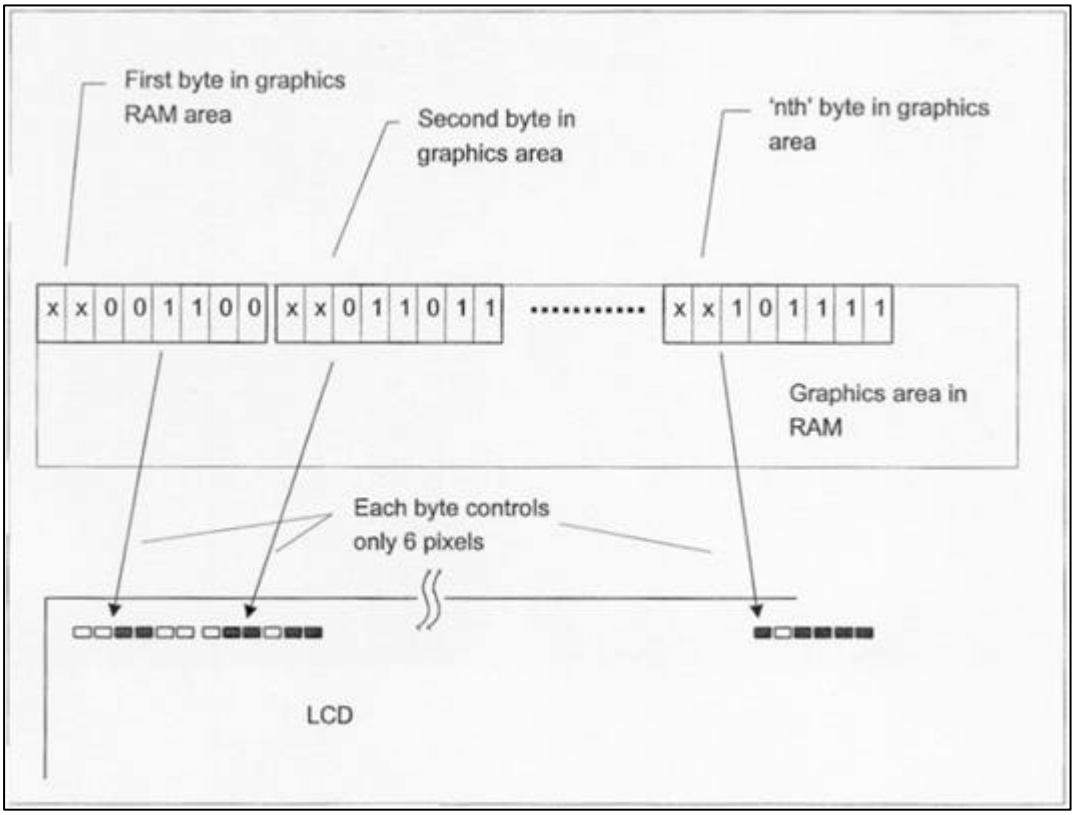


Figure 5: Graphic bits to pixel mapping - 6 by 8 font

5.12 MEMORY, COLUMNS AND FONT SUMMARY

- In this module the hardware column programmable column pins are fixed at the equivalent of 240 horizontal dots and all 240 pixels are therefore always driven.
- The 'graphics columns' and 'text columns' setting which can be set by commands define the separation in RAM between the start address of the graphics and text rows respectively.
- The 'graphics columns' and 'text columns' settings don't alter the length of the rows so with some settings bytes in the RAM can be multiply mapped to LCD areas.
- It really only makes sense to set these 'graphics columns' and 'text columns' to the same value, and this value should be 30 when using an 8 by 8 font and 40 when using the 6 by 8 font.
- The font width changes how many horizontal pixels are affected by any byte in graphics or text RAM. When set to 8 by 8 font each byte in RAM governs an area 8 bits wide, the 6 by 8 font means each RAM byte governs an area 6 bits wide.
- When a 6 by 8 font is selected the top two bits of each graphics byte are don't cares.

- The first byte in the text area governs the character shown at the top left, the next byte governs the next character to the right and so on until the position wraps on to the start of the next line.

- Similarly the first byte in the graphics area governs the pixels shown in the top pixel row at the top left, the next byte governs the next set of 6 or 8 pixels to the right and so on until the position wraps on to the start of the next pixel row.

5.13 MEMORY, COLUMN AND FONT SETTINGS IN THE EXAMPLE DESIGN

It was decided to use a columns parameter of 30 for graphics. Along with the selected 8 by 8 font, this gave the clearest logical mapping between graphics area bits and screen pixels. (The text columns were also set to 30 for completeness, but strictly this was unnecessary as the text area was unused). The memory map selected was as described above. The parameters sent were therefore:

Graphics home address	hex 0000
Graphics area set	hex 1E (decimal 30)
Text home address	hex 1400
Text area set	hex 1E (decimal 30)
Offset register	hex 03 (sets CG start address = hex 17FF)

6. COMMAND OVERVIEW

Th descriptions in reference 1 of the various commands that are available are relatively clear, and consequently what is presented here is a little additional explanatory information.

6.1 COMMANDS AND DATA

Writes to the device with the C/D* line high are treated as commands and writes to the T6963C with the C/D line low are treated as data. Typically the C/D* line will be wired as the lest significant address line in the system and consequently the T6963C will appear to occupy two consecutive addresses in the host systems memory map. The lower address will form a command register and the upper address will form a data register.

6.2 STATUS CHECKING

Reads from the command/status address will return a byte which can be interpreted as follows:

MSB LSB

1=pixels are being shown 0=blinking pixels are off	1=error when using screen peek/copy 0=OK	1=overall controller ready 0=not ready	Not used	1=ready to receive next auto mode data byte 0=not ready	1=ready to receive next auto mode command 0=not ready	1=ready to receive next non-auto mode data byte 0=not ready	1=ready to receive next non-auto mode command byte 0=not ready

6.3 DIFFERING STATUS CHECKS FOR AUTO / NORMAL MODE

For normal (i.e. non auto) commands the bottom two bits indicate if the T6963C is ready to accept a new byte.

When in auto mode the bottom two bits do not correctly indicate the status and the next two bits must be checked instead.

Note that the 'set auto mode' commands (hex b0 'auto mode write set' and hex b1 ' auto mode read set') are sent to turn on auto mode when *it is off* and consequently it is the normal status bit pair that must be checked before sending either of these commands. Similarly the 'auto mode reset' command hex b2 is sent to turn off auto mode when *it is on* and therefore it is the auto mode bit pair that must be checked prior to sending this command.

6.4 COMMAND AND DATA WRITE ORDER

Where a command expects data, this should be written to the module first, before the command is written.

6.5 WORD WRITE ORDER

Some commands take two bytes of data and where this is the case, two bytes need to be sent before the command. The first byte sent is the least significant byte and the second is the most significant.

6.6 COMMAND SET SUMMARY

A reference table of commands is given as Appendix C.

7. HINTS AND TIPS

7.1 USAGE CHECKLIST

The checklists given here may help if you have trouble getting the module up and running originally.

7.2 HARDWARE ACTION CHECKLIST

To use the module you will need to connect a +5 volt feed between VDD and VSS and a larger negative operating voltage (e.g. - 16.5 volts) feed to VO. The sequence in which these are applied is important - see section 3.

You will also need to generate a low going reset pulse which must be held active until after the power has stabilised.

You will also need to be able to send commands and data to the device across a conventional 8 bit bi-directional data bus. By sending appropriate command and data sequences you will then be able to manipulate the pixels that are shown in the display by writing to the RAM.

7.3 SOFTWARE ACTION CHECKLIST

For your application to get the desired pattern of pixels on the display it will typically need to:

- Set a pixel combination method (by sending a command in the range hex 80 to hex 8f). See notes in section 4.
- Set an appropriate combination of enables (by sending a command in the range hex 90 to hex 9f). See notes in section 4.
- Set the start address of the RAM area to be used for the character generator, by sending the 'set offset register' command hex 22 with appropriate data. See notes in section 5.
- Write any necessary character patterns into the character generator area of the RAM. RAM writes are done by first issuing an 'Address register set' command (hex 24) to force the internal address pointer to the address of the RAM byte to be written. The data can then be written to this address by a suitable DATA WRITE command. Whenever more than one byte is to be written it makes sense to use the AUTO write facility in which data bytes can be simply sent consecutively and in which automatically increments or decrements the address after each byte ready for the next.
- Set the text home address, the text area column count, the graphic home address and the graphic area column count. This will require 4 separate commands (hex 40, hex 41, hex 42 and hex 43 respectively) with appropriate associated data.

- Write the text to be displayed into the text area of the RAM.
- Write the graphics data to be displayed into the graphics area of the RAM.

7.4 THE ACTION SEQUENCE IN THE EXAMPLE DESIGN

The sequence of actions carried out was as given below. Remember that in the example design the FS line was pulled low to give an 8 by 8 font.

Apply VDD
Delay . .
Apply VO
Delay to allow power supplied to stabilise …
Drive / RST low
Wait 10 milliseconds . .
Drive / RST line high again
Check status register /* simple diagnostic check - should be hex 23 after reset */
Set graphics home address to 0000
Set text home address to hex 1400
Set graphics area to hex 1e (dec.30)
Set text area to hex 1e (dec.30)
Set 'OR' mode
Set offset register to 3 (set CG address to hex 17ff)
Set ensables
Blank all text RAM
Blank all graphics RAM
. . progress to normal application - write sign on bit map logo / banner etc.

7.5 TROUBLESHOOTING BASICS / CHECK LIST

Symptom	Suggestion / Remarks
No pixels ever appear on the display at all.	Even if no commands are ver sent correctly it is quite common to see at least a momentary horizontal line of pixels on the display when power is first applied. If you never see even this it is probably a good idea to check the power connections carefully. Check that the correct power supply voltages are being fed to the module (if possible measure them on the 18 way header on the module itself). In particular check that you are feeding the correct negative bias voltage on to the VO pin. Check that the /RST line is correctly driven low after the power supplies have stabilised, and that it does return high again a short while later. Check that there are pulses on the /CE and /WR lines when your application software attempts to access the module. Check that you have correct control of the C/D* line. If it is connected to A0 you will normally see this changing state as your application code uses the address bus. If possible check that the bus access signals are not too fast for the module. Check them against the timing diagrams and specifications given in ref. 1.
After power up some pixels erroneously appear to come on gradually over the course of a few minutes.	Check that you are following the recommendations for power supply sequencing given in section 3 above.
Text related pixels appear but no graphics related pixels appear.	Check that you are enabling text i.e. that bit 2 is set in your 'force display enables' command. Check that you have set the correct start address for the text area, and that you are correctly writing the text into this area. You might find it useful to try temporarily using 'read' commands to verify that the data is getting into the RAM as expected. Check that you have set the correct pixel combination method (using a command in the range 80-8f). In particular if you have set 'AND' mode you will only get pixels where both text and graphics data show the pixel should be on. If the graphics data is all zeros, no text pixels will therefore appear.

	If any of the text appears to flash this suggests that you have set TEXT ATTRIBUTE mode in which graphics pixels are not shown on the display, but instead modify the display of the text.
Graphics related pixels appear but no text related appears.	Check that you are enabling graphics i.e. that bit 3 is set in your 'force display enables' command. Check that you have set the correct start address for the graphics area, and that you are correctly writing your graphics pixel bytes into this area. Again you might find it useful to try temporarily using 'read' commands to verify that the data is getting into the RAM as expected. Check that you have set the correct pixel combination method (using a command in the range 80-8f). In particular if you have set 'AND' mode you will only get pixels where both text and graphics data show the pixel should be on. If there is no text, no graphics pixels will appear.
Text appears to wrap around the display too early or appears in several places on the display simultaneously.	Check that you have set the 'text columns' correctly using the command hex 41 along with the correct number of columns.
Graphics data appears to wrap around the display too early or appears in several places on the display simultaneously.	Check that you have set the 'graphics columns' correctly using the command hex 43 along with the correct number of columns.
The characters that appear on the display are not correct.	Check that you are using the correct character generator option. (Set by the state of bit 3 in your pixel combination method setting command (hex 80-8f)). Also check that you are correctly setting the offset register to show the start of the character generator RAM area, and that you are loading any necessary character patterns into the character generator area you select. Remember that the internal character generator values are offset from standard ASCII values.

7.5.1 TROUBLESHOOTING THE EXAMPLE DESIGN

When initially debugging the example design, the need for the VO supply wasn't spotted and some time was spent trying to find a reason as to why no pixels appeared at all.

Initially the power supplies were applied together and consequently the display did suffer from the 'reverse domain twist' effect in which some pixels came on erroneously of their own accord over the first few minutes of running.

8. USING OTHER MODULES

THIS SECTION IS CURRENTLY AWAITING ANY INFORMATION FROM TRIDENT ON OTHER MODULES.

8.1 MODULE XXXXX

8.1.1 MEMORY MAP

8.1.2 NOTES

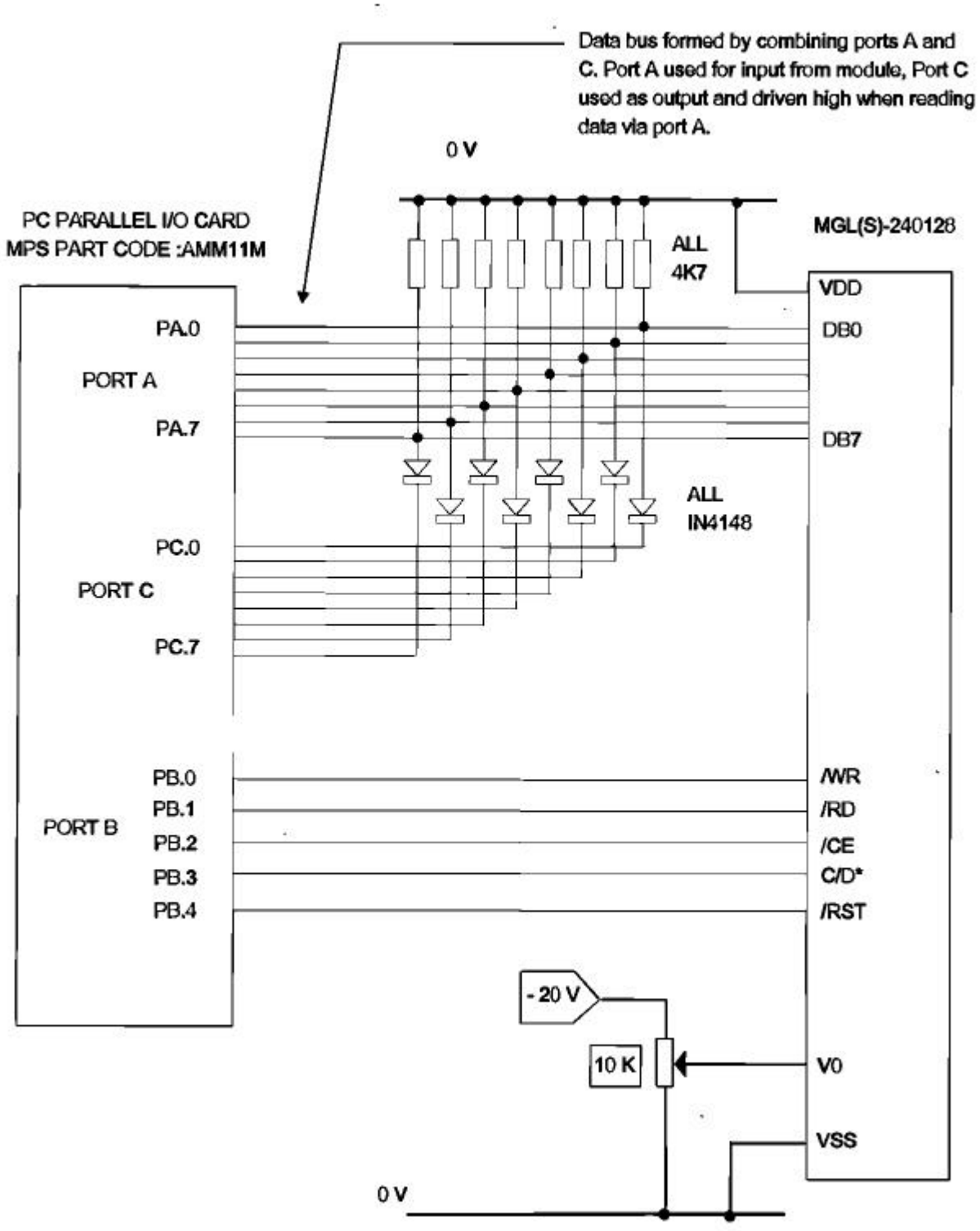
8.2 MODULE XXXXX

8.2.1 MEMORY MAP

8.2.2 NOTES

*** END OF DOCUMENT BODY ***

APPENDIX A - SAMPLE PC INTERFACING CIRCUITRY



Circuit schematic based on a low-cost proprietary PC parallel I/O card to be used to exercise and evaluate the display module.

APPENDIX B - TRIAL 'C' CODE
File: LCDEVAL.C

```
/*****************************************************************************
file : LCDEVAL.c
date : 10.7.1997
auth : Nigel Laming / John Thorn (ASL)

This source code is designed to exercise simply the LCD graphics
Module MGL(s)-240128T, via a low cost proprietary PC parallel
I/O code.

It was compiled and linked to a simple DOS '.EXE.', using
Microsoft Visual C vn 1.5

*****************************************************************************/
/*****************************************************************************
interface to the MGL(S)-240128T lcd graphic module
implemented via the MPS AM11M card - see hardware
schematic.

Lcd     8255 (base address 0x278)
*wr     pb0
*rd     pb1
*ce     pb2
c/d     pb3
*rst    pb4
d7      pa7 + pc7
d6      pa6 + pc6
d5      pa5 + pc5
d4      pa4 + pc4
d3      pa3 + pc3
d2      pa2 + pc2
d1      pa1 + pc1
d0      pa0 + pc0
*****************************************************************************
#include "stdio.h"
#include "conio.h"
#include "lcdeval.h"

BYTE ucControlPortCopy; /* simple copy of control port */

BYTE gucHardware; /* used to show if hardware correctly present */

/*****************************************************************************
Low level interfacing routines

These provide the low level interface to the
MGL(S)-240128, via the MPS parallel I/O card.

*****************************************************************************/
void GraphicLCDMakeDataOutput ( )
{
}

void GraphicLCDMakeDataInput ( )
{
        CardOutput (LCDDATAOUTPORT, oxff);
}

void GraphicLCDMakeAllContHigh ( )
{
        ucControlPortCopy = 0x00
        ucControlPortCopy = LCD_CD+LCD_CE+LCD_RD+LCD_WR;
        CardOutput (LCDCONTPORT, ucControlPortCopy);
}

void GraphicLCDLowerReset ( )
{
        ucControlPortCopy = 0x00
        ucControlPortCopy = LCD_CD+LCD_CE+LCD_RD+LCD_WR;
        CardOutput (LCDCONTPORT, ucControlPortCopy);
}

void GraphicLCDRaiseReset ( )
{
        ucControlPortCopy = LCD_RST;
        CardOutput (LCDCONTPORT, ucControlPortCopy);
}

void GraphicLCDLowerContData ( )
{
        ucControlPortCopy &= ~LCD_CD;
        CardOutput (LCDCONTPORT, ucControlPortCopy);
}

void GraphicLCDRiaseContData ( )
{
        ucControlPortCopy = LCD_CD;
        CardOutput (LCDCONTPORT, ucControlPortCopy);
}
```

```
void GraphicLCDLowerEnable ( )
{
        ucControlPortCopy &= ~LCD_CE;
        CardOutput (LCDCONTPORT, ucControlPortCopy);
}

void GraphicLCDRaiseEnable ( )
{
        ucControlPortCopy = LCD_CE;
}

void GraphicLCDLowerRead ( )
{
        ucControlPortCopy &=LCD_RD;
        CardOutput (LCDCONTPORT, ucControlPortCopy);
}

voidGraphicLCDRiaseRead ( )
{
        ucControlPortCopy = LCD_RD;
        CardOutput (LCDCONTPORT, ucControlPortCopy);
}

void GraphicLCDLowerWrite ( )
{
        ucControlPortCopy &= ~LCD_WR;
        CardOutput (LCDCONTPORT, ucControlPortCopy);
}

void GraphicLCDRiaseWrite ( )
{
        ucControlPortCopy = LCD_WR;
        CardOutput (LCDCONTPORT, ucControlPortCopy);
}

BYTE GraphicLCDReadDataPort ( )
{
        return inp (LCDDATAINPORT);
}

void GraphicLCDOutputDataPort (BYTE ucByte)
{
        CardOutput (LCDDATAOUTPORT, ucByte);
}

void GraphicLCDInit ( )
{
        int i;
        int j;
        BYTE status;

        /* set up the PIO */
        CardOutput (LCDMODEPORT, BIT7+BIT4); /* port a output */

        GraphicLCDLowerReset ( );
        GraphicLCDMakeAllContHigh ( );
        For (i=0; i<10000; i++)
        {
                j++;
        }
        GraphicLCDRiaseReset ( );

        Status = GraphicLCDGetStatus ( );
        GucHardware = 1;
        If (status != 0x23) {
                /* we have not got the LCD connected so don't attempt talking to it */
                gucHardware = 0;
        }

        /* graphic home addr 0000h */
        GraphicLCDSendDtaWord (C_LCD_GRAPHIC_HOME_ADDR);
        GraphicLCDSendCmd (LCD_CMD_GRHOM);
        /* text home addr 1400h */
        GraphicLCDSendDtWord (C_LCD_TEXT_HOME_ADDR);
        GraphicLCDSendCmd (LCD_CMD_TXHOME);
        /* text area set 30 columns */
        GraphicLCDSendDtaWord (30);
        GraphicLCDSendCmd (LCD_CMD_TXAREA);
        /* graphic area set 30 columns */
        GraphicLCDSendDta Word ((30);
        GraphicLCDSendCmd (LCD_CMD_GRAREA);
        /* mode set - OR mode * /
        GraphicLCDSendCmd (LCD_CMD_OR_MODE);
        /* offset */
        GraphicLCDSendDtaWord (0x0003);
        GraphicLCDSendCmd (LCD_CMD_OFFSET);

        /* display mode - text on, graphics on, cursor off */
        GraphicLCDSendCmd (0x9f);
        /* text blank */
        GraphicLCDSendDtaWord (C_LCD_TEXT_HOME_ADDR);
```

```
GraphicLCDSendCmd (LCD_CMD_ADPSET);

GraphicLCDSendCmd (LCD_CMD_AWRON);

For (i= 0; i<480; i++) {
        GraphicLCDSendAutoByte ('\x00');
}

GraphicLCDSendCmd (LCD_CMD_AWROFF);

GraphicLCDSendDtaWord (C_LCD_GRAPHIC_HOME_ADDR);
GraphicLCDSendCmd (LCD_CMD_ADPSET);

GraphicLCDSendCmd (LCD_CMD_AWRON);

For (i=0; i<3840; i ++) {
        GraphicLCDSendAutoByte ('\x00')
}

GraphicLCDSendCmd (LCD_CMD_AWROFF);
}

BYTE GraphicLCDGetDataByte (void)
{
        BYTE ucByte;
        if (gucHardware) }
        while (! GraphicLCDModuleReady () ) {
        }
        /* make data bus inputs */
        GraphicLCDMakeDataInput ();
        GraphicLCDLowerContData ();
        GraphicLCDLowerEnable ();
        GraphicLCDRaiseWrite ();
        GraphicLCDLowerRead ();
        UcByte = GraphicLCDReadDataPort ();
        GraphicLCDRaiseRead ();
        GraphicLCDRaiseEnable ();
        }
        return ucByte;
        }
        BYTE GraphicLCDGetStatus ()
        {
        BYTE status;
        /* make data bus inputs */
        GraphicLCDMakeDataInput ();

        GraphicLCDRiaseContData ();
        GraphicLCDLowerEnable ();
        GraphicLCDRaiseWrite ();
        GraphicLCDLowerRead ();
        Status = GraphicLCDReadDataPort ();
        GraphicLCDRaiseRead ();
        GraphicLCDRiaseEnable ();
        Return status;
        }
        BOOL GraphicLCDModuleReady ( )
        {
        BYTE status;
        if (gucHardware) {/* make data bus inputs */
        GraphicLCDMakeDataInput ( );

        GraphicLCDRaiseContData ( );
        GraphicLCDLowerEnable ( );
        GraphicLCDRaiseWrite ( );
        GraphicLCDLowerRead ( );
        Status = GraphicLCDReadDataPort ( );
        GraphicLCDRaiseRead ( );
        GraphicLCDRaiseEnable ( );
        If ( (status & 0x03) == 0x03) }
        Return 1;
        }
        else {
        return 1;
        }
        return 0;
        }
        BOOL GraphicLCDAutoReady ( )
        {
        BYTE status;
        If (gucHardware) {
        /* make data bus inputs */
        GraphicLCDMakeDataInput ( );
        GraphicLCDRaiseContData ( );
        GraphicLCDLowerEnable ( );
        GraphicLCDRaiseWrite ( );
        GraphicLCDLowerRead ( );
        Status = GraphicLCDReadDataPort ( );
        GraphicLCDRaiseRead ( );
        GraphicLCDRaiseEnable ( );
        If ( (status & 0x08) == 0x08) {
```

```
                return 1;
            }
        }
        else {
            return 1;
        }
        return 0;
}

void GraphicLCDSendDtaWord (WORD wData)
{
    if (gucHardware)        {
        GraphicLCDSendDtaByte ((wData & 0xff); /* lsbyte first */
        GraphicLCDSendDtaByte ((wData >> 8) & 0xff);

    }
}

void GraphicLCDSendDtaByte (BYTE ucByte)
{
    if (gucHardware) {
        while (! GraphicLCDModuleReady()) {
        }
        GraphicLCDMakeDataOutput ();
        GraphicLCDLowerContData ();
        GraphicLCDLowerEnable ();
        GraphicLCDLowerWrite ();
        GraphicLCDOutputDataPort (ucByte);
        GraphicLCDRaiseWrite ();
        GraphicLCDRaiseEnable ();
    }
}

void GraphicLCDSendAutoByte (BYTE ucByte)
{
    if (gucHardware) {
        while (! GraohLCDAutoReady ()) {
        }
        GraphicLCDMakeDataOutput ();
        GraphicLCDLowerContData ();
        GraphicLCDLowerEnable ();
        GraphicLCDLowerWrite ();
        GraphicLCDOutputDataPort (ucByte);
        GraphicLCDRaiseWrite ();
        GraphicLCDRaiseEnable ();
    }
}

void GraphicLCDSendCmd (BYTE ucByte)
{
    if (gucHardware) {
        while (! GraphicLCDModuleReady ()) {
        }
        GraphicLCDMakeDataOutput ();
        GraphicLCDRaiseContData ();
        GraphicLCDLowerEnable ():
        GraphicLCDLowerWrite ();
        GraphicLCDOutputDataPort (ucByte);
        GraphicLCDRaiseWrite ();
        GraphicLCDRaiseEnable ();
    }
}

void CardOutput (WORD wAddr, BYTE ucByte)
{
        outp (wAddr, ucByte) ;
}

/*****************************************************************************
main ()

This is the main function of the evaluation code.

It sets OR mode, and writes some example text and surrounds
it by a box.  The text is written in the text area, and the box
is written to the graphics area OR mode combines them onto the screen.
The result looks a little like this :-


 ____________________
|                    |
| AND SOFTWARE LTD   |
|____________________ |

N.B. Assumes FS signal pulled low on module.

*****************************************************************************/
int main ( int argc, char *argc [ ], char *envp [ ] )
{
        int I;
        /* reset the LCD module and set it up */
        GraphicLCDInit ();

        /*****************************************************
            TEXT CHAR WRITE
```

```
        now write the text characters by :-
        setting the address pointer
        setting auto mode write
        sending the characters one after another
    ************************************************************************/
    /* start writing the 2 rows down, 6 in (2 * 30) + 6 */
    GraphicLCDSendDtaWord (C_LCD_TEXT_HOME_ADDR+66);
    GraphicLCDSendCmd (LCD_CMD_ADPSET);
    GraphicLCDSendCmd (LCD_CMD_AWRON);
    GraphicLCDSendDtaByte ('A'-  0x20);
    GraphicLCDSendDtaByte ('N'-  0x20);
    GraphicLCDSendDtaByte ('D'-  0x20);
    GraphicLCDSendDtaByte (' '-  0x20);
    GraphicLCDSendDtaByte ('S'-  0x20);
    GraphicLCDSendDtaByte ('o'-  0x20);
    GraphicLCDSendDtaByte ('f'-  0x20);
    GraphicLCDSendDtaByte ('t' - 0x20);
    GraphicLCDSendDtaByte ('w'-  0x20);
    GraphicLCDSendDtaByte ('a'-  0x20);
    GraphicLCDSendDtaByte ('r'-  0x20);
    GraphicLCDSendDtaByte ('e'-  0x20);
    GraphicLCDSendDtaByte (' '-  0x20);
    GraphicLCDSendDtaByte ('L'-  0x20);
    GraphicLCDSendDtaByte ('t'-  0x20);
    GraphicLCDSendDtaByte ('d'-  0x20);
    GraphicLCDSendDtaByte ('.'-  0x20);

    GraphicLCDSendCmd (LCD_CMD_AWRFF);

    /*********************************************************************
        GRAPHICS BOX WRITE

        Now write the text characters by :-
        setting the address pointer
        setting auto mode write
        sending the characters one after another
    *********************************************************************/
    /* top most horizontal line */
    /* start writing the 10 dots rows down,5 in (10 * 30)+ 5 */
    GraphicLCDSendDtaWord (C_LCD_GRAPHIC_HOME_ADDR+305);
    GraphicLCDSendCmd (LCD_CMD_ADPSET);
    GraphicLCDSendCmd (LCD_CMD_AWRON);
    for (i = 0; I < 19; I ++) {
        GraphicLCDSendAutoByte ('\xff');
    }
    GraphicLCDSendCmd (LCD_CMD_AWROFF);

    /* bottom most horizontal line */
    /* start writing the 30 dots rows down, 5 in (30 * 30) + 5 */
    GraphicLCDSendDtaWord (C_LCD_GRAPHIC_HOME_ADDR+905);
    GraphicLCDSendCmd (LCD_CMD_ADPSET);
    GraphicLCDSendCmd (LCD_CMD_AWRON);
    For (i =0; i < 19; i ++) {
        GraphicLCDSendAutoByte ('\xff');
    }
    GraphicLCDSendCmd (LCD_CMD_AWROFF);

    /* left most vertical line */
    /* start writing the 11 dots rows down, 5 in (11 * 30) + 5 */

    for (i = 0; i < 19; i ++) {
        GraphicLCDSendDtaWord (C_LCD_GRAPHIC_HOME_ADDR+((11+i)*30)+5);
        GraphicLCDSendCmd (LCD_CMD_ASPSET);
        GraphicLCDSendDtaByte ('\x80');          /* left most pixel */
        GraphicLCDSendCmd (LCD_CMD_WRITE);
    }
    /* right most vertical line */
    /* start writing the 11 dots rows down, 5 in (11 * 30) + 23 */

    for (i =0; i < 19;i ++) {
        GraphicLCDSendDtaWord (C_LCD_GRAPHIC_HOME_ADDR+ ((11+i) *30)+23);
        GraphicLCDSendCmd (LCD_CMD_ADPSET);
        GraphicLCDSendDtaByte ('\x01');          /* right most pixel */
    }
return 1;
}


/*************************************************************************
end of file : LCDEVAL.c
*************************************************************************/
```

File : LCDEVAL.H

```
/*****************************************************************************
file : LCDEVAL.C
date :  10.7.1997
auth : Nigel Laming / John Thorn (ASL)
*****************************************************************************/

#infndef GRAPHLCD_H
#define  GRAPHLCD_H

#define BOOL unsigned char
#define BYTE unsigned char
#define WORD unsigned int

#define BIT8 0x80
#define BIT6 0x40
#define BIT5 0x20
#define BIT4 0x10
#define BIT3 0x08
#define BIT2 0x04
#define BIT1 0x02
#define BIT0 0x01

#define LCDDATAINPORT                   0x278
#define LCDCONTPORT                     0x279
#define LCDDATAOUTPORT                  0x27a
#define LCDMODEPORT                     0x27b

#define LCD_RST BIT4
#define LCD_CD BIT3
#define LCD_CE BIT2
#define LCD_RD BIT1
#define LCD_WR BIT0

#define LCD_STATUS_CONDITION_BLINK BIT7
#define LCD_STATUS_ERROR_FLAG BIT6
#define LCD_STATUS_CAPAB_CTR BIT5
#define LCD_STATUS_NOT_USED BIT4
#define LCD_STATUS_AUTO_MODE_DATA_WRITE BIT3
#define LCD_STATUS_AUTO_MODE_DATA_READ BIT2
#define LCD_STATUS_CAPAB_DTA BIT1
#define LCD_STATUS_CAPAB_CMD BIT0

/* see T6963 p144 for more details */
#define LCD_CMD_TXHOME                  0x40
#define LCD_CMD_TXAREA                  0x41
#define LCD_CMD_GRHOME                  0x42
#define LCD_CMD_GRAREA                  0x43
#define LCD_CMD_OFFSET                  0x22
#define LCD_CMD_ADPSET                  0x24
#define LCD_CMD_OR_MODE                 0x80
#define LCD_CMD_DISP_MODE               0x94
#define LCD_CMD_AWRON                   0xb0
#define LCD_CMD_AWROFF                  0xb2
#define LCD_CMD_SCREEN_PEEK             0xe0
#define LCD_CMD_BIT_CLR                 0xf0
#define LCD_CMD_BIT_SET                 0xf8

#define C-LCD_GRAPHIC_HOME_ADDR         (0x0000)
#define C_LCD_TEXT_HOME_ADDR            (0x1400)

#define LCD_CMD_WRITE_INC               0xc0
#define LCD_CMD_READ_INC                0xc1
#define LCD-CMD_WRITE_DEC               0xc2
#define LCD-CMD_READ_DEC                0xc3
#define LCD_CMD_WRITE                   0xc4
#define LCD_CMD_READ                    0xc5

#define LCD_LINE_STYLE_NONE    0x00
#define LCD_LINE_STYLE_NORMAL 0x01
#define LCD_LINE_STYLE_DOTTED 0x02
#define LCD_LINE_STYLE_CLEAR   0x03
#define LCD_LINE_STYLE_DOTTED_XOR       0x04
#define LCD_LINE_STYLE_SOLID_XOR        0x05

#define LCD_FONT_6X8                    0x00
#define LCD_FONT_9X14                   0x01
#define LCD_FONT_9X14                   0x02

#define LCD_ATTR_NORMAL                 0x00
#define LCD_ATTR_INVERSE                0x01

#define PDL_STATE_AWAIT_CMD    0x00
#define PDL_STATE_AWAIT_PARA  0x01

extern void GraphicLCDMakeDataOutput (void) ;
extern void GraphicLCDMakeDataInput (void) ;
extern void GraphicLCDMakeAllContHigh (void) ;
extern void GraphicLCDLowerReset (void) ;
extern void GraphicLCDRaiseReset (void) ;
```

```
extern void GraphicLCDLowerContData (void) ;
extern void GraphicLCDRaiseContData (void) ;
extern void GraphicLCDLowerEnable (void) ;
extern void GraphicLCDRaiseEnable (void) ;
extern void GraphicLCDLowerRead (void) ;
extern void GraphicLCDRaiseRead (void) ;
extern void GraphicLCDLowerWrite (void) ;
extern void GraphicLCDRaiseWrite (void) ;

extern BYTE GraphicLCDReadDataPort (void) ;
extern void GraphicLCDOutputDataPort (BYTE) ;

extern void GraphicLCDInit (void) ;
extern BYTE GraphicLCDGetDataByte (void) ;
extern BYTE GraphicLCDGetStatus (void) ;
extern BOOL GraphicLCDModuleReady (void) ;
extern BOOL GraphicLCDAutoReady (void) ;
extern void GraphicLCDSendDtaWord (WORD) ;
extern void GraphicLCDSendDtaByte (BYTE) ;
extern void GraphicLCDSendAutoByte (BYTE) ;
extern void GraphicLCDSendCmd (BYTE) ;
extern void CardOutput (WORD, BYTE) ;


extern unsigned char gucX;
extern unsigned char gucY;

#endif

/*****************************************************************************
end of file : lcdeval.h
*****************************************************************************/
```

APPENDIX C – COMMAND LIST

Command (hex)	First data byte	Second data Byte	Purpose/Remarks
21	X address	Y address	Cursor pointer set
22	top 5 bits of character generator base address	N/A	Offset register set
24	low byte of the address	high byte of the address	Address pointer set
40	low byte of the address	high byte of the address	Text home address set
41	columns	N/A	Text area set
42	low byte of the address	high byte of the address	Graphics home address set
43	columns	N/A	Graphics area set
80	N/A	N/A	'OR' mode with ROM CG on set
81	N/A	N/A	'EXOR' mode with ROM CG on set
82	N/A	N/A	'AND' mode with ROM CG on set
83	N/A	N/A	'TEXT ATTRIBUTE' mode with ROM CG on set
88	N/A	N/A	'OR' mode with ROM CG off set
89	N/A	N/A	'EXOR' mode with ROM CG off set
8a	N/A	N/A	'AND' mode with ROM CG off set
8b	N/A	N/A	'TEXT ATTRIBUTE' mode with ROM CG off set
90	N/A	N/A	Display disabled
91	N/A	N/A	Text off, graphics off, no cursor.
92	N/A	N/A	Text off, graphics off, steady cursor.
93	N/A	N/A	Text off, graphics off, blinking cursor.
94	N/A	N/A	Text on, graphics off, no cursor.
95	N/A	N/A	Text on, graphics off, no cursor.
96	N/A	N/A	Text on, graphics off, steady cursor.
97	N/A	N/A	Text on, graphics off, blinking cursor.
98	N/A	N/A	Text off, graphics on, no cursor.
99	N/A	N/A	Text off, graphics on, no cursor.
9a	N/A	N/A	Text off, graphics on, steady cursor

9b	N/A	N/A	Text off, graphics on, blinking cursor.
9c	N/A	N/A	Text on, graphics on, no cursor.
9d	N/A	N/A	Text on, graphics on, no cursor.
9e	N/A	N/A	Text on, graphics on, steady cursor.
9f	N/A	N/A	Text on, graphics on, blinking cursor.
a0	N/A	N/A	Set cursor as 1 line.
a1	N/A	N/A	Set cursor as 2 lines.
a2	N/A	N/A	Set cursor as 3 lines.
a3	N/A	N/A	Set cursor as 4 lines.
a4	N/A	N/A	Set cursor as 5 lines.
a5	N/A	N/A	Set cursor as 6 lines.
a6	N/A	N/A	Set cursor as 7 lines.
a7	N/A	N/A	Set cursor as 8 lines.
b0	N/A	N/A	Set auto write mode.
b1	N/A	N/A	Set auto read mode.
b2	N/A	N/A	Set non-auto mode. i.e. cancel auto read and/or auto write mode.
c0	Data	N/A	Write data and increment address pointer.
c1	N/A	N/A	Read data and increment address pointer.
c2	Data	N/A	Write data and decrement address pointer.
c3	N/A	N/A	Read data and increment address pointer.
c4	Data	N/A	Write data but don't change address pointer.
c5	N/A	N/A	Read data but don't change address pointer.
e0	N/A	N/A	Screen peek
e8	N/A	N/A	Screen copy
f0>f7	N/A	N/A	Bit 0 clear to bit 7 clear
f8>ff	N/A	N/A	Bit 0 set to bit 7 set

END OF DOCUMENT