Application Note, V1.0, January 2008

AP16132

XE167

Controlling a 20 x 4 character display with semi-graphical functions with a XE167 device.

Microcontrollers

Never stop thinking

Edition 2008-01

Published by
Infineon Technologies AG
81726 Munich, Germany

AP16132
Revision History: 2008-01 V1.0
Previous Version: none

Page	Subjects (major changes since last revision)

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:
mcdocu.comments@infineon.com

Table of Contents **Page**

1 Introduction

This document explains how to interface a character-based 20x4 display with semi graphical functionalities to an XE167 microcontroller using some I/O ports.

In this document the TM204 display module will be used: this device is a common cheap display module; it is produced by many manufacturers.

The TM204 module includes a built-in controller that simplifies driving the display, moreover it integrates an LED (or CCFL) backlight and the LCD color may vary according to the manufactures specifications.

Driving this display module requires no particular hardware, only few ports lines will be used, so you can apply the code presented in this application note to an alternative Infineon microcontroller.

The scope of this application is to completely control the TM204 features such as strings printing, simple graphics drawing and building simple animations: these functionalities can be used for a very wide range of applications beginning from the visualization of various parameters of machinery to a smart debugging system of source code, showing the status of the program.

In the following chapters first the hardware involved in this application note is described and then the software programming; finally you'll find some conclusions.

Required background: Basics of peripherals interfacing.

2 The hardware

This chapter describes the display module used in this application note and how to connect it to the XE167 microcontroller.

2.1 The TM204 display module with the HD44780 controller

The TM204 module is a very widespread display module, it exists in versions built by various manufacturers that differ in some details like the size, the LCD color and the backlight type. The characteristics that match all these products is the built-in controller: it's the HD44780 or a clone product like the KS0066, the S6A0069, the SED1278, the ST7066 or the SPLC780A1; the protocol used to communicate to the HD44780 display has become a standard in the past years and now it is the most common controller used with character displays.

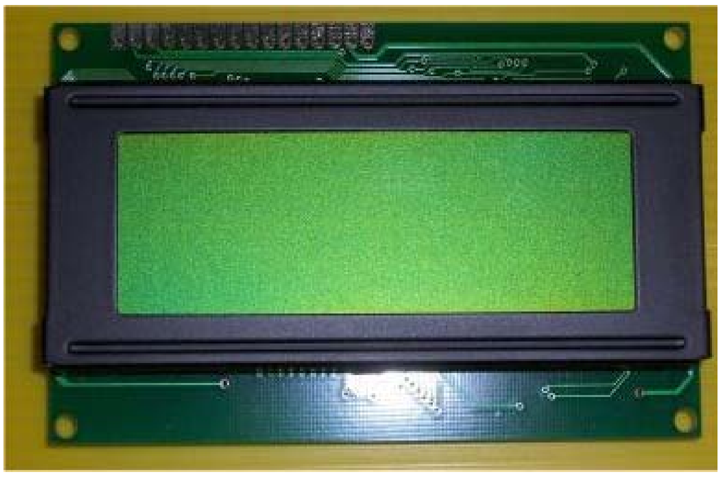


Figure 1 The TM204 display module

The TM204 module allows the display of 20 characters per line on 4 different lines: this format is supported by the HD44780 controller as well as other common formats such 8x2, 16x1, 16x2, 16x4, 20x2 or 24x2. The display is so called semi-graphical because every character is composed by a matrix of 5x8 dots; the controller has a built-in character map but it allows you to redefine some of these characters by editing the dot matrix: with these personalized symbols you can create some graphical effects or you can make your strings more impressive.

Therefore, controlling the behavior of the TM204 display module means controlling the HD44780 controller: it communicate with the XE167 through a data bus and a control bus; the supported interface is the so called "Motorola style" interface with a Read/Write signal (RW) and an Enable command (E). Data flows toward the controller on an 8 bits data bus (but a 4 bits data bus could be used too) in the form of commands and it can be read back too. Using commands to read and write data on the display avoids the need to use any kind of address bus, only one line is reserved: the RS line (that corresponds to the A0 line). This defines if the actual communication must be processed by the controller as a command or as a data: for example, writing a string corresponds to send the "write a char" command and sends the characters to write.

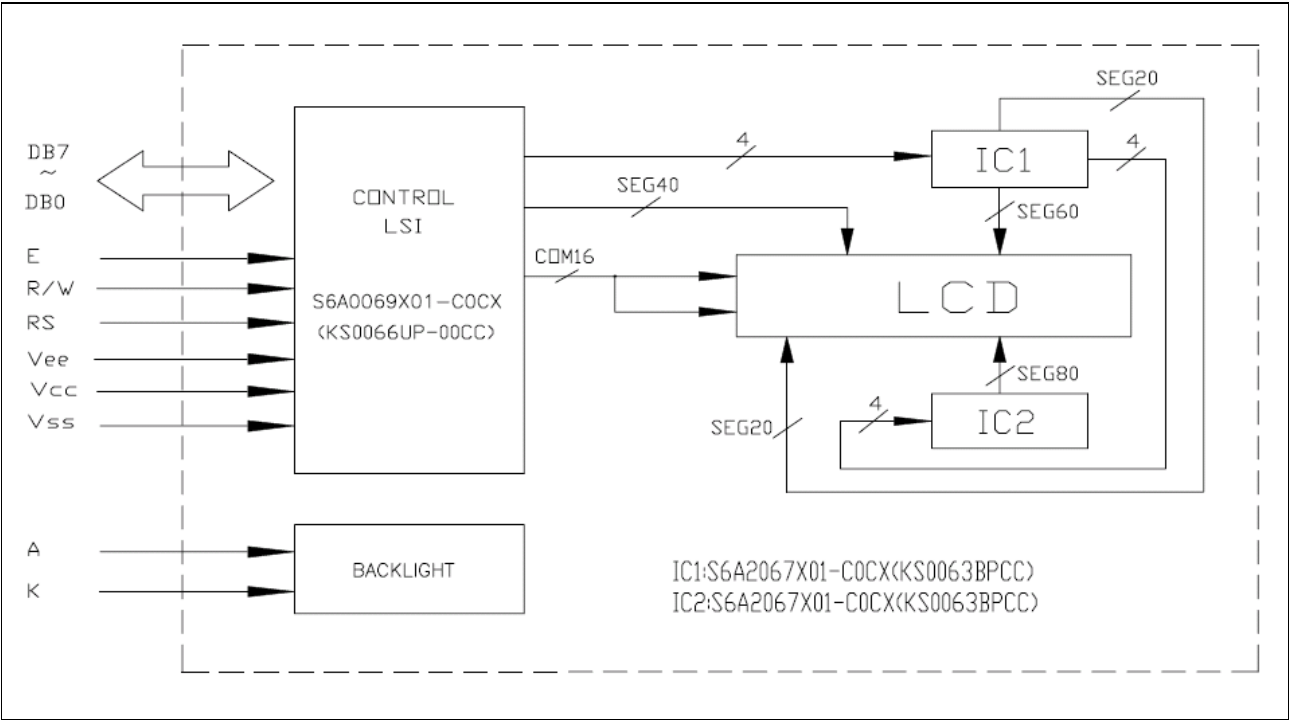
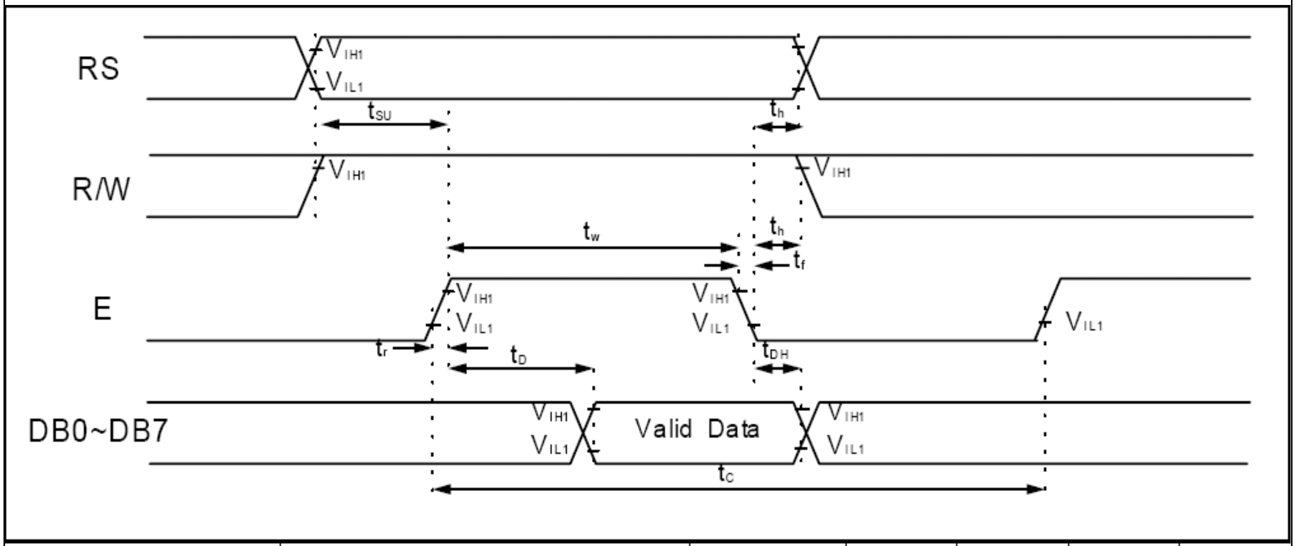


Figure 2 Block scheme of the TM204 module

In this application note the control signals necessary to communicate with the HD44780 controller will be generated on the pins of the I/O ports: this procedure could be called "interface emulation" and it is the easier way realize the communication.

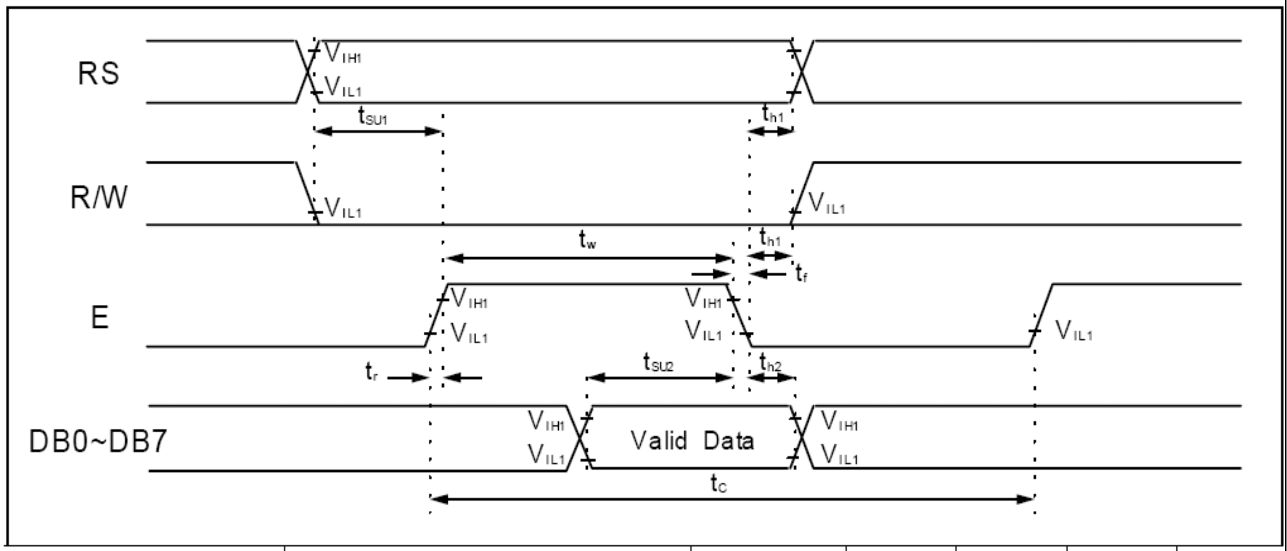
An other way to do this is to connect the display module as a peripheral (or as a RAM bank) through the EBC of the XE167 microcontroller: in this case the addressing of the HD44780 requires at least two memory locations of 8 bits.

The next tables, taken from the HD44780 datasheet, show what are the time requirements for data and commands reading/writing from/to the controller.



Read Mode	E Cycle Time	tc	500	-	-	ns
	E Rise / Fall Time	t_R,t_F	-	-	20	
	E Pulse Width (High, Low)	tw	230	-	-	
	R/W and RS Setup Time	tsu	40	-	-	
	R/W and RS Hold Time	t_H	10	-	-	
	Data Output Delay Time	t_D	-	-	120	
	Data Hold Time	t_{DH}	5	-	-	

Figure 3 HD44780 reading timings



Write Mode	E Cycle Time	tc	500	-	-	ns
	E Rise / Fall Time	t_R,t_F	-	-	20	
	E Pulse Width (High, Low)	tw	230	-	-	
	R/W and RS Setup Time	tsu1	40	-	-	
	R/W and RS Hold Time	t_{H1}	10	-	-	
	Data Setup Time	tsu2	80	-	-	
	Data Hold Time	t_{H2}	10	-	-	

Figure 4 HD44780 writing timings

The other aspects that concern the communications between the HD44780 controller and the XE167 such as how to print a string to the display and how to build a simple animation will be described in the Software chapter.

As seen before, the TM204 module is equipped with an LED display backlight; typically it is powered on or off depending on the specific application requirements but it could be a good idea to enhance your application with an automatic backlight ignition according to the actual lighting conditions: this could be done simply using a photodiode that drives the LED backlight directly or sampling the photodiode status with the XE167 microcontroller and then driving the backlight separately; this second approach is preferable because it gives you more control on the backlight status, for example in this case it is possible to enforce a software hysteresis on the backlight ignition to avoid it useless stress or it is possible to drive the display module with different graphics according with the lighting conditions or you can implement some particular politics of power saving.

Another important issue is how to feed the power supply of the LCD: normally a display module is equipped with two different supply lines, one of these is dedicated to supply the controller (called logic part supply) and it requires a stable 5 Volts voltage, the other one is related to the supply of the LCD panel: to vary this voltage value leads to a contrast adjustment of the display, this is useful to compensate a temperature variation or a different angle of view. The best choice for this kind of display is to connect a trimmer between the 5 Volts and the ground reference and feed the LCD part supply with the variable voltage that exits from the wiper so you can adjust the contrast manually.

The next figure shows the complete setup that has been just described: note the amplification stage of the photodiode that matches its voltage levels with the microcontroller parallel port ones; with this configuration it isn’t necessary to acquire the photodiode voltage status with an analog to digital converter, the built-in trigger present on every I/O pin of the XE167 microcontroller returns a logic level high or low when a specific voltage threshold has been crossed so the logic status of this input pin indicates if the environmental lighting condition is light or darkness.

After processing, the lighting status exits from another microcontroller's pin and activates a transistor used as a switch that feeds directly the display backlight: the amount of current that flows toward the backlight LEDs could be regulated with another trimmer used to set the display backlight brightness.

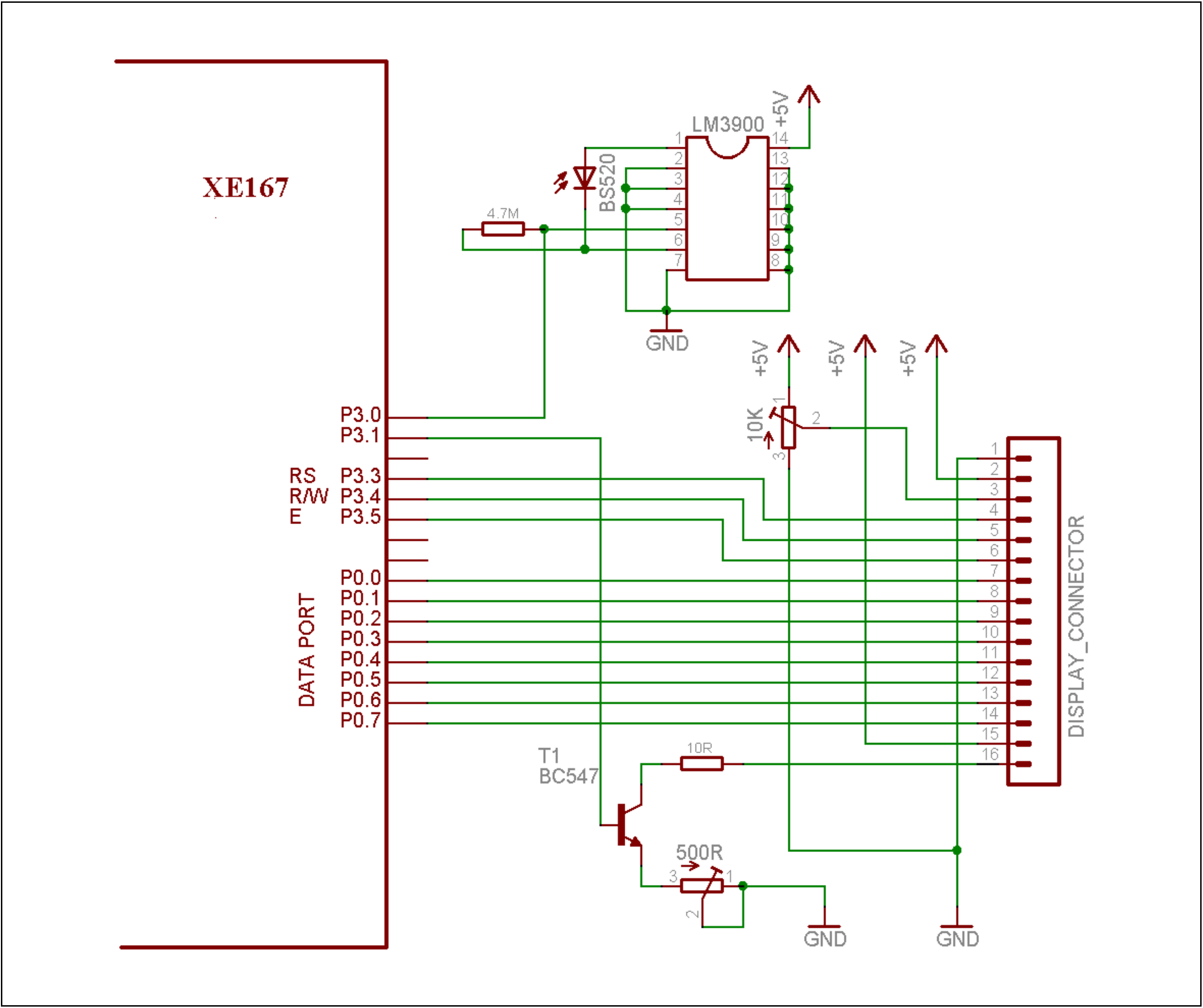


Figure 5 Connections diagram

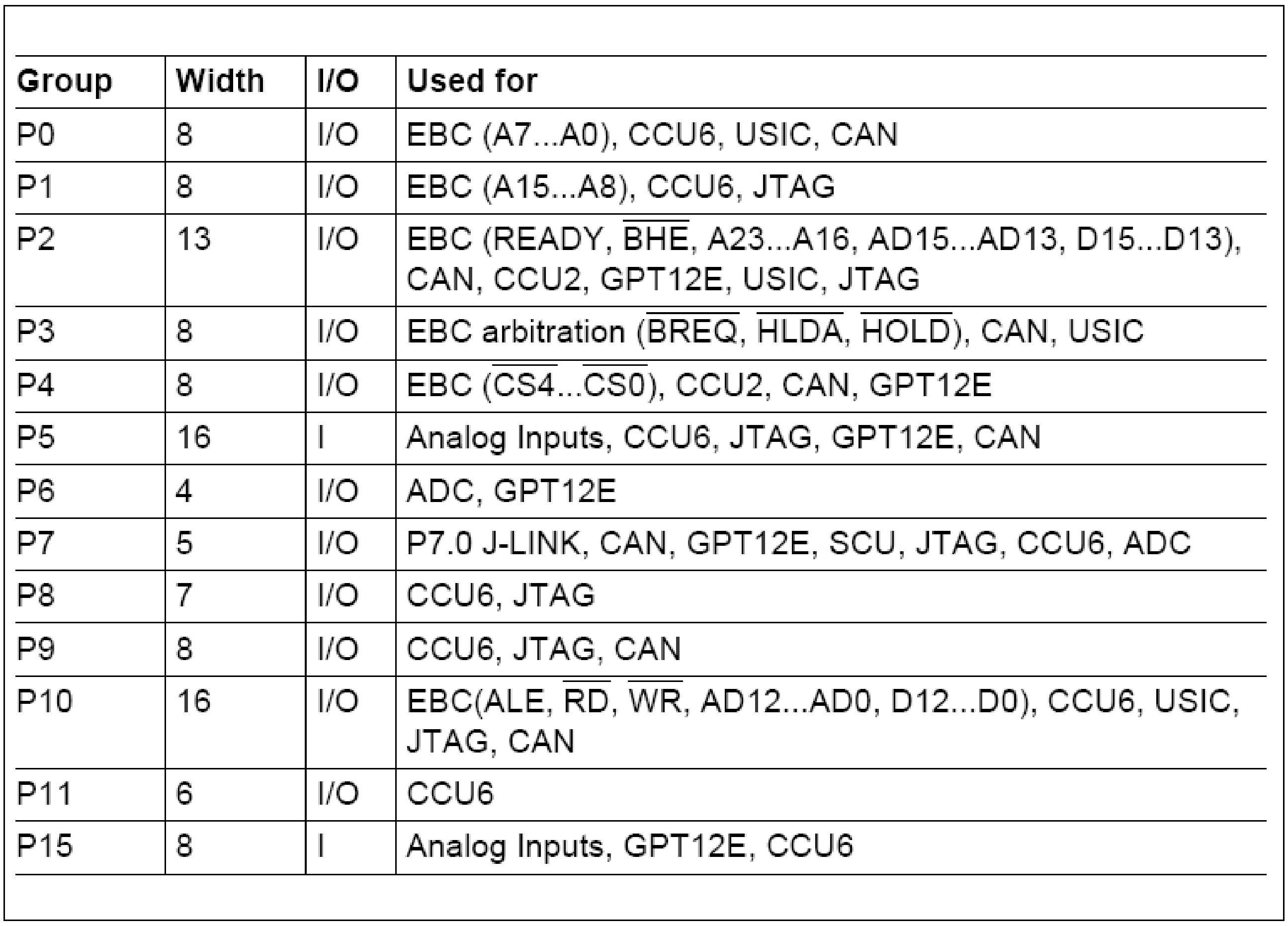
2.2 The XE167's parallel ports

Before starting to communicate with the display module it's important to know the characteristic of the XE167's I/O lines: this high level microcontroller is has with 116 I/O lines grouped into 13 ports (named P0…P11 and P15) with some important features:

- Bit-addressability;
- Tri-stated input mode;
- Push-pull or open drain output mode;
- Programmable port driver control;
- I/O voltage range from 3.0 Volts to 5.5 Volts.

These ports have different width and they could be grouped by further advanced features, for example P5 and P15 are analog input ports, some ports have the selectable input threshold capability and others have a power saving ability.

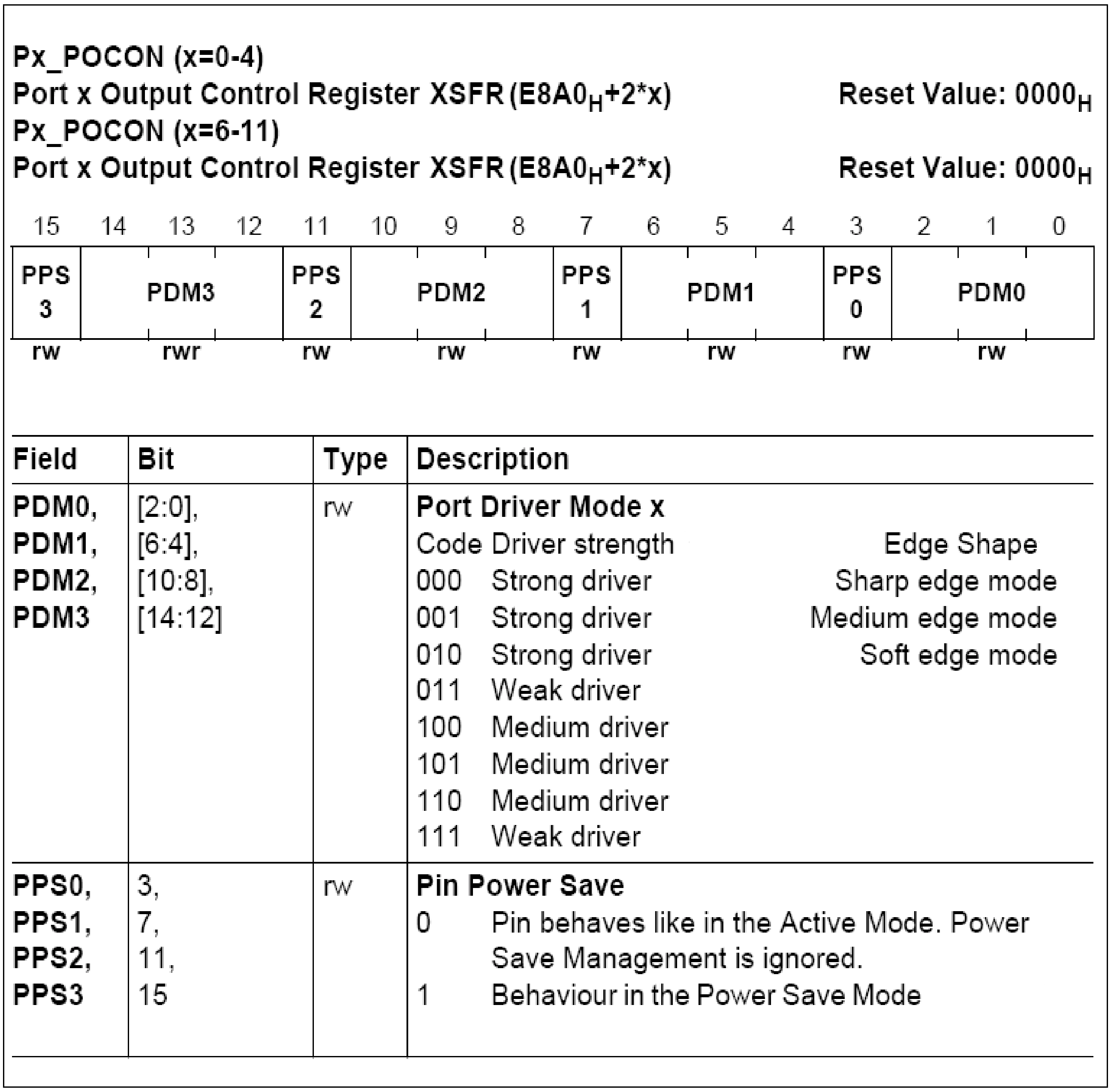
Since the XE167 microcontroller has a lot of built-in peripherals with a large amount of external signals, every I/O pin has more than one functionality: typically a pin can be used as input for two different peripheral modules or as an output for three of them (called alternate functions), it could be configured to be a specific EBC or JTAG signal or a simple general purpose I/O pin.



Group	Width	I/O	Used for
P0	8	I/O	EBC (A7...A0), CCU6, USIC, CAN
P1	8	I/O	EBC (A15...A8), CCU6, JTAG
P2	13	I/O	EBC (READY, $\overline{\text{BHE}}$, A23...A16, AD15...AD13, D15...D13), CAN, CCU2, GPT12E, USIC, JTAG
P3	8	I/O	EBC arbitration ($\overline{\text{BREQ}}$, $\overline{\text{HLDA}}$, $\overline{\text{HOLD}}$), CAN, USIC
P4	8	I/O	EBC ($\overline{\text{CS4}}$...$\overline{\text{CS0}}$), CCU2, CAN, GPT12E
P5	16	I	Analog Inputs, CCU6, JTAG, GPT12E, CAN
P6	4	I/O	ADC, GPT12E
P7	5	I/O	P7.0 J-LINK, CAN, GPT12E, SCU, JTAG, CCU6, ADC
P8	7	I/O	CCU6, JTAG
P9	8	I/O	CCU6, JTAG, CAN
P10	16	I/O	EBC(ALE, $\overline{\text{RD}}$, $\overline{\text{WR}}$, AD12...AD0, D12...D0), CCU6, USIC, JTAG, CAN
P11	6	I/O	CCU6
P15	8	I	Analog Inputs, GPT12E, CCU6

Figure 6 Summary of the port functionalities

All the possible settings just described are conveniently configed by a small set of registers; they will be presented next.



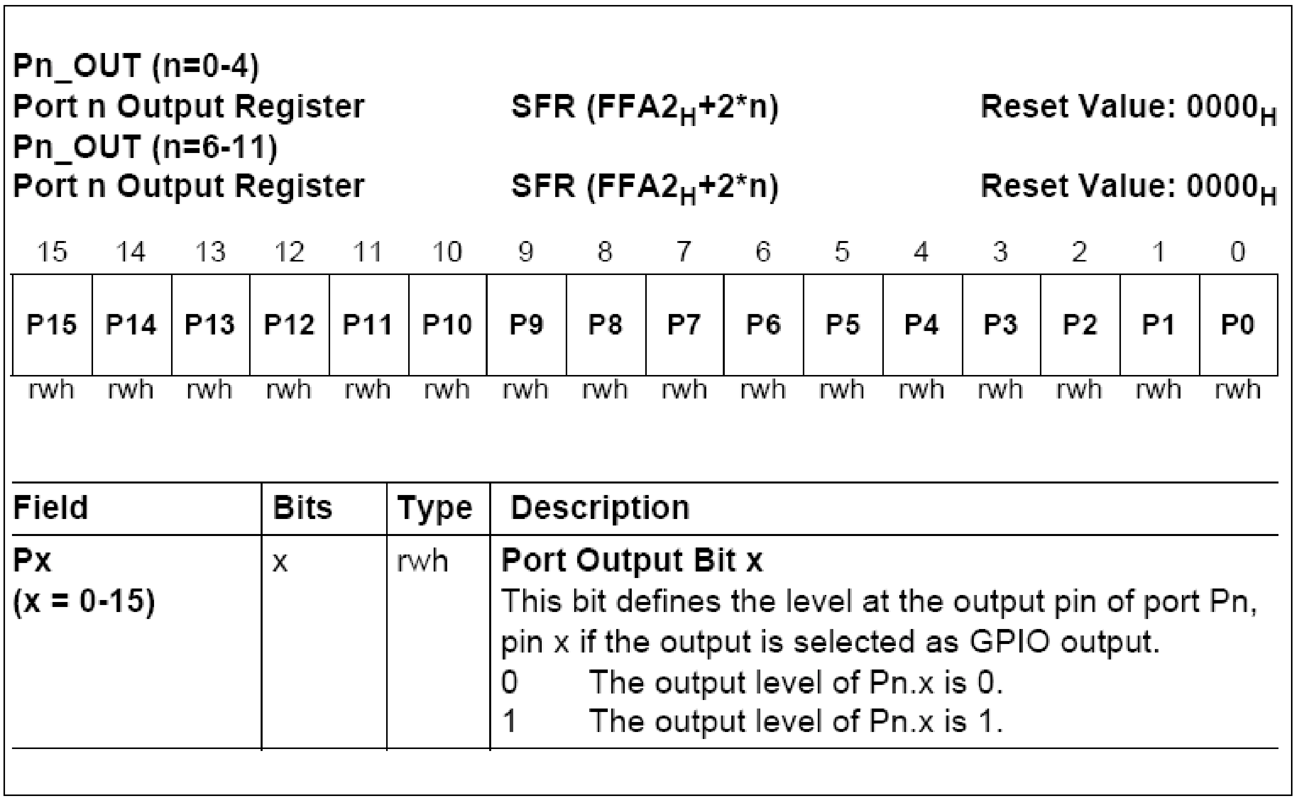
Px_POCON (x=0-4)
Port x Output Control Register XSFR (E8A0$_H$+2*x) **Reset Value: 0000$_H$**
Px_POCON (x=6-11)
Port x Output Control Register XSFR (E8A0$_H$+2*x) **Reset Value: 0000$_H$**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PPS 3	PDM3			PPS 2	PDM2			PPS 1	PDM1			PPS 0	PDM0		
rw	rwr			rw	rw			rw	rw			rw	rw		

Field	Bit	Type	Description
PDM0, PDM1, PDM2, PDM3	[2:0], [6:4], [10:8], [14:12]	rw	**Port Driver Mode x** Code Driver strength — Edge Shape 000 Strong driver — Sharp edge mode 001 Strong driver — Medium edge mode 010 Strong driver — Soft edge mode 011 Weak driver 100 Medium driver 101 Medium driver 110 Medium driver 111 Weak driver
PPS0, PPS1, PPS2, PPS3	3, 7, 11, 15	rw	**Pin Power Save** 0 Pin behaves like in the Active Mode. Power Save Management is ignored. 1 Behaviour in the Power Save Mode

Figure 7 The Px_POCON registers

The POCON registers define the port driver strength and the slew rate; the selection is done in groups of four pins and it can be done for every I/O port.



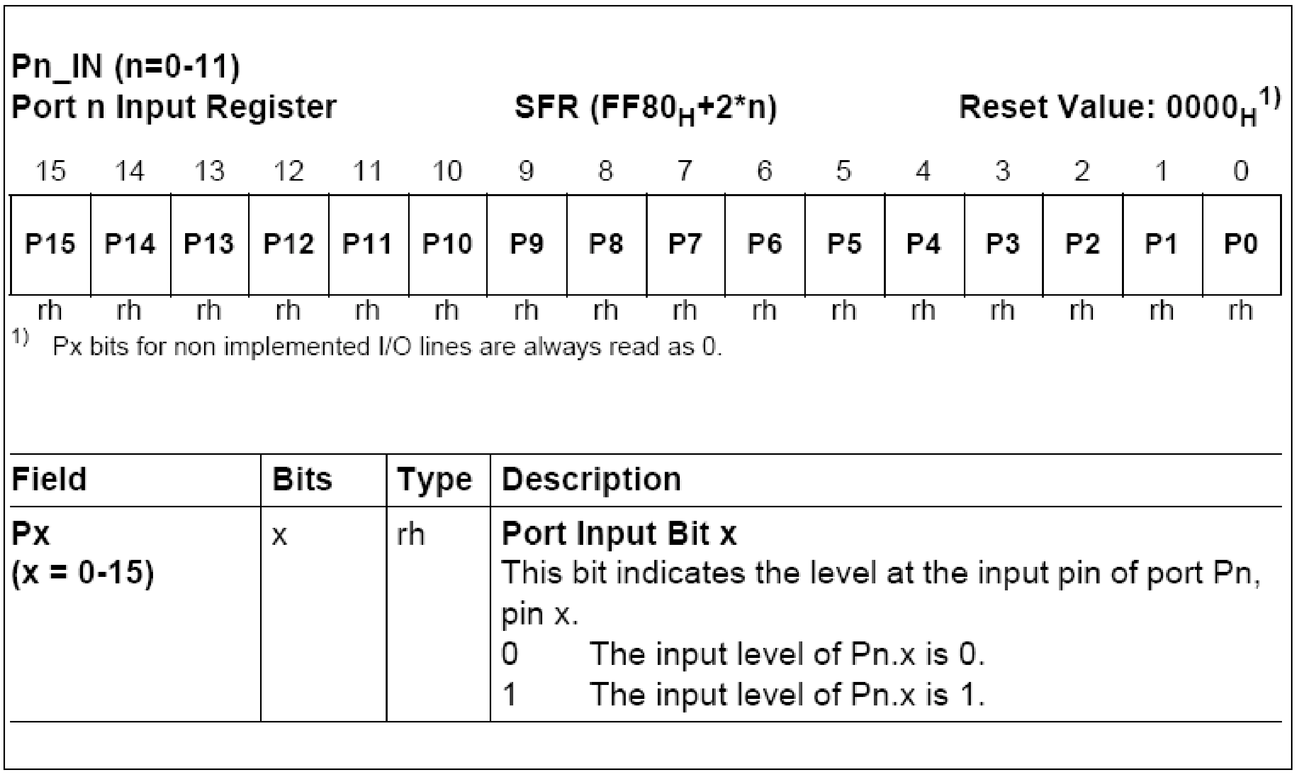
Pn_OUT (n=0-4)
Port n Output Register **SFR ($FFA2_H$+2*n)** **Reset Value: 0000_H**
Pn_OUT (n=6-11)
Port n Output Register **SFR ($FFA2_H$+2*n)** **Reset Value: 0000_H**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P15	**P14**	**P13**	**P12**	**P11**	**P10**	**P9**	**P8**	**P7**	**P6**	**P5**	**P4**	**P3**	**P2**	**P1**	**P0**
rwh	rwh	rwh	rwh	rwh	rwh	rwh	rwh	rwh	rwh	rwh	rwh	rwh	rwh	rwh	rwh

Field	Bits	Type	Description
Px **(x = 0-15)**	x	rwh	**Port Output Bit x** This bit defines the level at the output pin of port Pn, pin x if the output is selected as GPIO output. 0 The output level of Pn.x is 0. 1 The output level of Pn.x is 1.

Figure 8 The Pn_OUT registers

When the ports are used in general purpose I/O mode the P_OUT register permit to set the logic voltage value present on the pin, there is one of this for every port.



Pn_IN (n=0-11)
Port n Input Register **SFR ($FF80_H$+2*n)** **Reset Value: 0000_H**[1)]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P15	**P14**	**P13**	**P12**	**P11**	**P10**	**P9**	**P8**	**P7**	**P6**	**P5**	**P4**	**P3**	**P2**	**P1**	**P0**
rh	rh	rh	rh	rh	rh	rh	rh	rh	rh	rh	rh	rh	rh	rh	rh

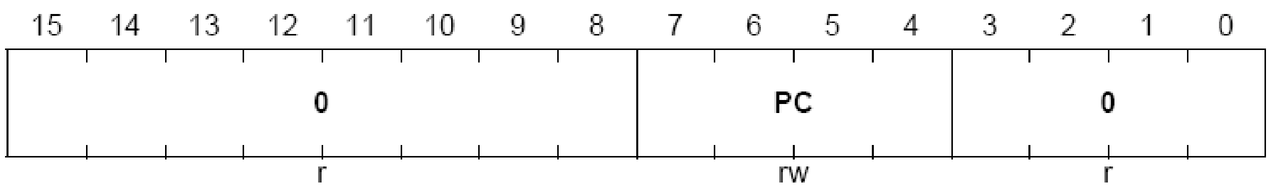
1) Px bits for non implemented I/O lines are always read as 0.

Field	Bits	Type	Description
Px **(x = 0-15)**	x	rh	**Port Input Bit x** This bit indicates the level at the input pin of port Pn, pin x. 0 The input level of Pn.x is 0. 1 The input level of Pn.x is 1.

Figure 9 The Pn_IN registers

The P_IN registers contain the values currently read at the input pins, also if a port line is assigned as output.

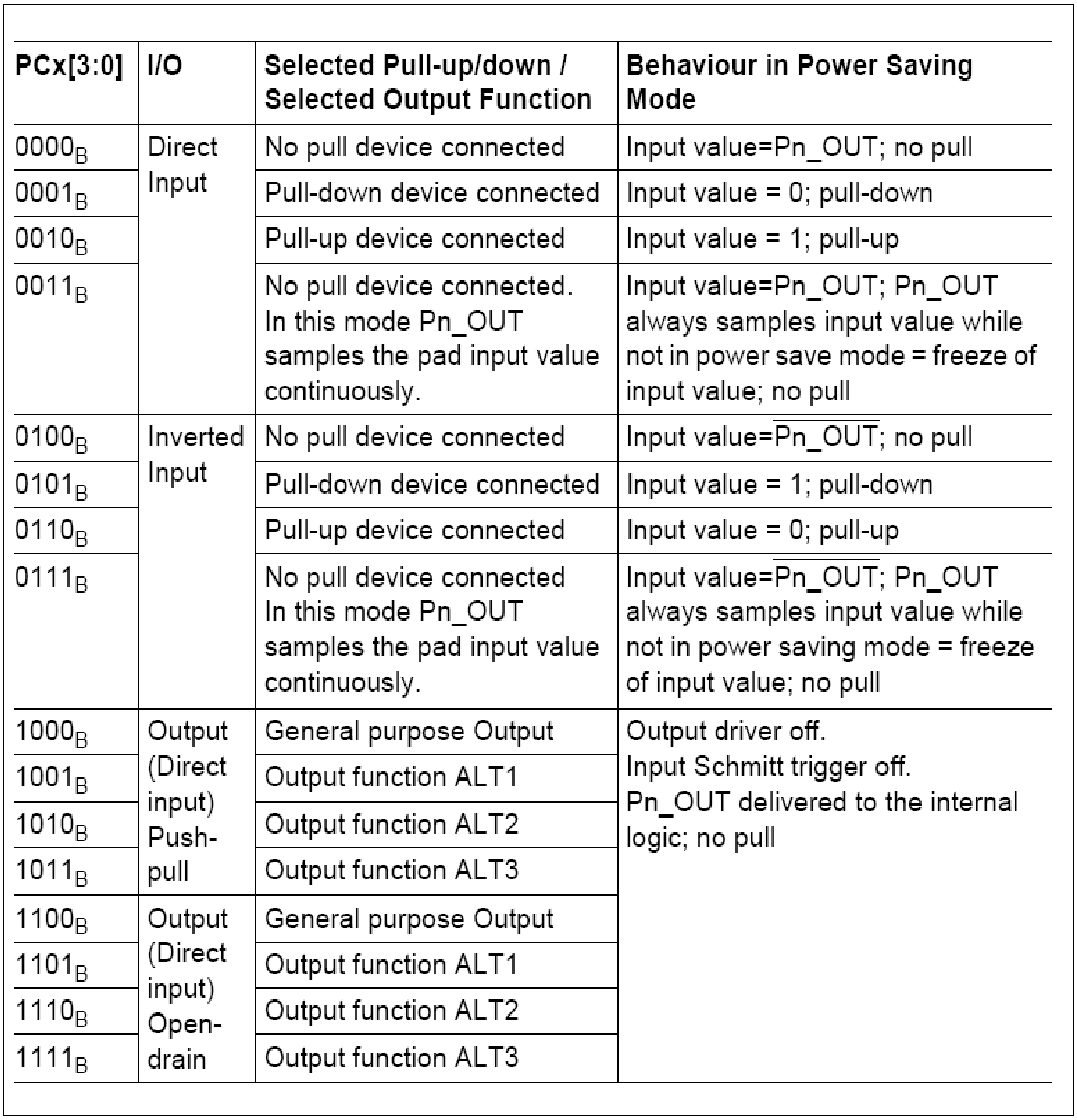
Pn_IOCRx (x=00-05)
Port n Input/Output Control Register x XSFR **Reset Value: 0000_H**



Field	Bits	Type	Description
PC	[7:4]	rw	**Port Input/Output Control Bit**
0	[3:0], [15:8]	r	reserved

Figure 10 The Pn_IOCRx registers

This group of registers selects the digital output and the input driver characteristic, such as pull-up/down devices, port direction, open-drain and alternate output selections; this can be done for each pin of the port separately. The PC field coding is explained in the next figure.



PCx[3:0]	I/O	Selected Pull-up/down / Selected Output Function	Behaviour in Power Saving Mode
0000_B	Direct Input	No pull device connected	Input value=Pn_OUT; no pull
0001_B		Pull-down device connected	Input value = 0; pull-down
0010_B		Pull-up device connected	Input value = 1; pull-up
0011_B		No pull device connected. In this mode Pn_OUT samples the pad input value continuously.	Input value=Pn_OUT; Pn_OUT always samples input value while not in power save mode = freeze of input value; no pull
0100_B	Inverted Input	No pull device connected	Input value=$\overline{\text{Pn\_OUT}}$; no pull
0101_B		Pull-down device connected	Input value = 1; pull-down
0110_B		Pull-up device connected	Input value = 0; pull-up
0111_B		No pull device connected In this mode Pn_OUT samples the pad input value continuously.	Input value=$\overline{\text{Pn\_OUT}}$; Pn_OUT always samples input value while not in power saving mode = freeze of input value; no pull
1000_B	Output (Direct input) Push-pull	General purpose Output	Output driver off. Input Schmitt trigger off. Pn_OUT delivered to the internal logic; no pull
1001_B		Output function ALT1	
1010_B		Output function ALT2	
1011_B		Output function ALT3	
1100_B	Output (Direct input) Open-drain	General purpose Output	
1101_B		Output function ALT1	
1110_B		Output function ALT2	
1111_B		Output function ALT3	

Figure 11 The I/O control register PC field

With this small set of registers we are able to completely control the status of the display connected to the XE167 microcontroller: this will be described in the next chapter.

The easistr way to prototype your application is to use an Infineon's Easy Kit application board: you can simply access the microcontrollers I/O pins and test your application in a very quick way.

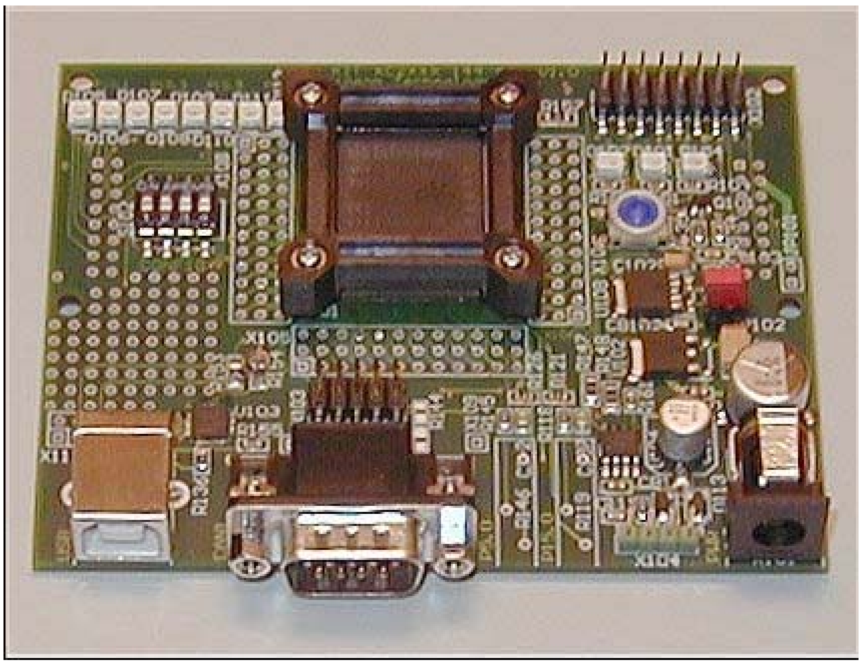


Figure 12 The “Easy kit” application board

3 The software

This chapter describes the software needed to configure the working display; it is divided into three logical layers that will be presented from the bottom to the top.

3.1 Low level communication procedures

Before you write something on the display module you need to know how to communicate with it: It's very important to understand this first basic step.

Looking at figures 3 and 4 you can see that the control signals involved in the communication process must be handled in a precise order and with specific timing requirements: this is the process called "interface emulation". The following table indicates the meaning of these signals.

RS	RW	Meaning
0	0	Write a command on data port
1	0	Write some data on data port
0	1	Read the internal status flag (busy flag)
1	1	Read data from the controller

Table 1 RS and RW control signals

In this way the logic levels present on these two lines leads to a different behavior of the HD44780 controller and it gives a different meaning to the data that will be present on the data bus. After these two lines have been correctly set, the E (enable) signal pulse triggers the selected operation on the data port; now, following this simple practice we have all the elements to build three basic procedures: LCDwriteCtrl, LCDwriteData and LCDreadData.

```
void LCDwriteCtrl(ubyte command)
{
wait60cycles();
DATA_PORT_OUT = command;
wait60cycles();
RW_LINE = 0;     // RW is low for the writing
wait60cycles();
RS_LINE = 0;     // RS low for commands
wait60cycles();
E_LINE = 1;      // E high
wait1ms();
E_LINE = 0;      // E low
wait60cycles();
}

void LCDwriteData(ubyte data)
{
DATA_PORT_OUT = data;
wait60cycles();
RW_LINE = 0;     // RW is low for the writing
wait60cycles();
RS_LINE = 1;     // RS is high for data
wait60cycles();
E_LINE = 1;
wait1ms();
E_LINE = 0;
wait60cycles();
}
```

```
ubyte LCDreadData(void)
{
ubyte retVal = 0;

DATA_PORT_CTRL0 = IN_PULL_UP        // set data port as input
DATA_PORT_CTRL1 = IN_PULL_UP;
DATA_PORT_CTRL2 = IN_PULL_UP;
DATA_PORT_CTRL3 = IN_PULL_UP;
DATA_PORT_CTRL4 = IN_PULL_UP;
DATA_PORT_CTRL5 = IN_PULL_UP;
DATA_PORT_CTRL6 = IN_PULL_UP;
DATA_PORT_CTRL7 = IN_PULL_UP;
wait60cycles();
RW_LINE = 1;
wait60cycles();
RS_LINE = 1;
wait60cycles();
E_LINE = 1;
wait1ms();
retVal = DATA_PORT_IN;
wait60cycles();
E_LINE = 0;
wait60cycles();
DATA_PORT_CTRL0 = OUT_GP_PUSH_PULL;   // restore data port as output
DATA_PORT_CTRL1 = OUT_GP_PUSH_PULL;
DATA_PORT_CTRL2 = OUT_GP_PUSH_PULL;
DATA_PORT_CTRL3 = OUT_GP_PUSH_PULL;
DATA_PORT_CTRL4 = OUT_GP_PUSH_PULL;
DATA_PORT_CTRL5 = OUT_GP_PUSH_PULL;
DATA_PORT_CTRL6 = OUT_GP_PUSH_PULL;
DATA_PORT_CTRL7 = OUT_GP_PUSH_PULL;
wait60cycles();
return retVal;
}
```

In these simple procedures you have to note the delay of 1 ms that permits to the display controller “to feel” the enable signal and than to trigger the selected operation: only 500 ns are strictly necessary but we use a delay of 1 ms to be sure that the operation is completely executed.

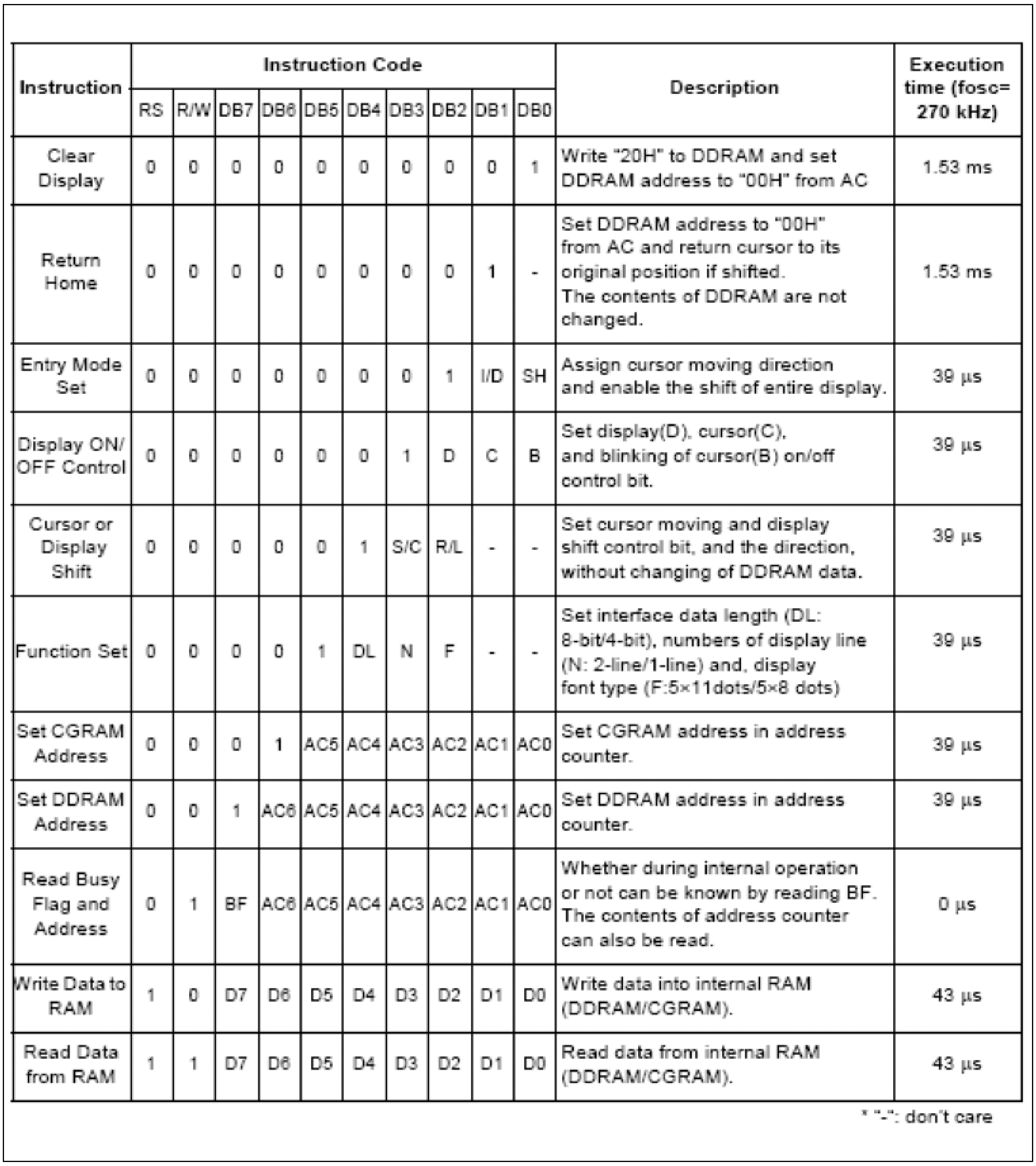
Moreover we have used another little delay inserted between the changes of the status of the control lines: this is not necessary but it is mandatory, especially for the high clock frequency operations, to wait for the electrical stabilization of the line: this could be important if the cable that connects the display module with the microcontroller is quite long.

Obviously the symbolic definitions used in these procedures refer to specific ports pins, in this example (see figure 5) we have used the followings:

```
#define RS_LINE         P3_OUT_P3
#define RW_LINE         P3_OUT_P4
#define E_LINE          P3_OUT_P5
#define DATA_PORT_OUT   P0_OUT
#define DATA_PORT_IN    P0_IN
#define DATA_PORT_CTRL0 P0_IOCR00
#define DATA_PORT_CTRL1 P0_IOCR01
#define DATA_PORT_CTRL2 P0_IOCR02
#define DATA_PORT_CTRL3 P0_IOCR03
#define DATA_PORT_CTRL4 P0_IOCR04
#define DATA_PORT_CTRL5 P0_IOCR05
#define DATA_PORT_CTRL6 P0_IOCR06
#define DATA_PORT_CTRL7 P0_IOCR07
#define RS_LINE_CTRL    P3_IOCR03
#define RW_LINE_CTRL    P3_IOCR04
#define E_LINE_CTRL     P3_IOCR05
```

3.2 Middle level operations

After we have seen how to communicate with the display controller we are ready to learn what commands it understands; the following figure shows you the complete list of the controller's command followed by a brief description.



Instruction	Instruction Code										Description	Execution time (fosc= 270 kHz)
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear Display	0	0	0	0	0	0	0	0	0	1	Write "20H" to DDRAM and set DDRAM address to "00H" from AC	1.53 ms
Return Home	0	0	0	0	0	0	0	0	1	-	Set DDRAM address to "00H" from AC and return cursor to its original position if shifted. The contents of DDRAM are not changed.	1.53 ms
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	SH	Assign cursor moving direction and enable the shift of entire display.	39 µs
Display ON/ OFF Control	0	0	0	0	0	0	1	D	C	B	Set display(D), cursor(C), and blinking of cursor(B) on/off control bit.	39 µs
Cursor or Display Shift	0	0	0	0	0	1	S/C	R/L	-	-	Set cursor moving and display shift control bit, and the direction, without changing of DDRAM data.	39 µs
Function Set	0	0	0	0	1	DL	N	F	-	-	Set interface data length (DL: 8-bit/4-bit), numbers of display line (N: 2-line/1-line) and, display font type (F:5×11dots/5×8 dots)	39 µs
Set CGRAM Address	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0	Set CGRAM address in address counter.	39 µs
Set DDRAM Address	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Set DDRAM address in address counter.	39 µs
Read Busy Flag and Address	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Whether during internal operation or not can be known by reading BF. The contents of address counter can also be read.	0 µs
Write Data to RAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Write data into internal RAM (DDRAM/CGRAM).	43 µs
Read Data from RAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Read data from internal RAM (DDRAM/CGRAM).	43 µs

* "-": don't care

Figure 13 Command set of the TM204 display module

The "clear display" and "return home" commands are trivial but the "entry mode set" is important to set the direction of the cursor, that has been moved automatically after every write operations, and to disable the shift of the entire screen. Another important command is the “display ON/OFF control” that permits to power on or to power off the display, to decide if you want to show the cursor on it and if this cursor has to blink or not.

The “cursor or display shift” command permits to select in what direction the cursor or the entire screen has to shift whereas the “function set” command allows you to configure your specific display module setting, its characters format, kind of interface and number of lines.

The “set CGRAM/DDRAM address” commands need to select a memory location to write to: the first of them refer to the internal character generator RAM while the seconds point directly to the displayed data memory. Other than these we have a command, “write data to RAM”, that permits to write the user data into the specific location selected with the previous commands and a “read data from RAM” that allows to read the memory content.

The last command we examine in figure 13 is the “read busy flag and address” command that gives you the possibility to poll the controller status.

Now, looking at this command set, you can understand the next procedures without any difficult.

```
void LCDinitialize(void)
{
DATA_PORT_CTRL0 = OUT_GP_PUSH_PULL;   //data port
DATA_PORT_CTRL1 = OUT_GP_PUSH_PULL;
DATA_PORT_CTRL2 = OUT_GP_PUSH_PULL;
DATA_PORT_CTRL3 = OUT_GP_PUSH_PULL;
DATA_PORT_CTRL4 = OUT_GP_PUSH_PULL;
DATA_PORT_CTRL5 = OUT_GP_PUSH_PULL;
DATA_PORT_CTRL6 = OUT_GP_PUSH_PULL;
DATA_PORT_CTRL7 = OUT_GP_PUSH_PULL;

RW_LINE_CTRL = OUT_GP_PUSH_PULL;      //RW    0 --> write ; 1 --> read
E_LINE_CTRL = OUT_GP_PUSH_PULL;       //E     active low
RS_LINE_CTRL = OUT_GP_PUSH_PULL;      //RS    0 --> commands ; 1 --> data


DATA_PORT_OUT = 0x00;
RS_LINE = 0;
RW_LINE = 0;
E_LINE = 1;

wait10ms();
wait10ms();
LCDwriteCtrl(0x38); //function set
wait10ms();
LCDwriteCtrl(0x38); //function set
wait1ms();
LCDwriteCtrl(0x08); //disp off
wait1ms();
LCDwriteCtrl(0x06); //entry mode set
wait1ms();
LCDwriteCtrl(0x01); //display clear
wait10ms();
LCDwriteCtrl(0x0c); //disp on
wait1ms();
}

void LCDdispON(void)
{
LCDwriteCtrl(0x0c); //disp on
wait1ms();
}

void LCDdispOFF(void)
{
LCDwriteCtrl(0x08); //disp off
wait1ms();
}
```

```
void LCDwriteChar(sbyte pchar)
{
LCDwriteData(pchar);
}

void LCDwriteString(sbyte *string)
{
uword inx=0;

while(string[inx] != '\0')
   {
   LCDwriteChar(string[inx]);
   inx++;
   }
}
void LCDhome(void)
{
LCDwriteCtrl(0x02);  //send clear and return home command
wait10ms();
}

void LCDclear(void)
{
LCDwriteCtrl(0x01);  //send clear command
wait10ms();
}
```

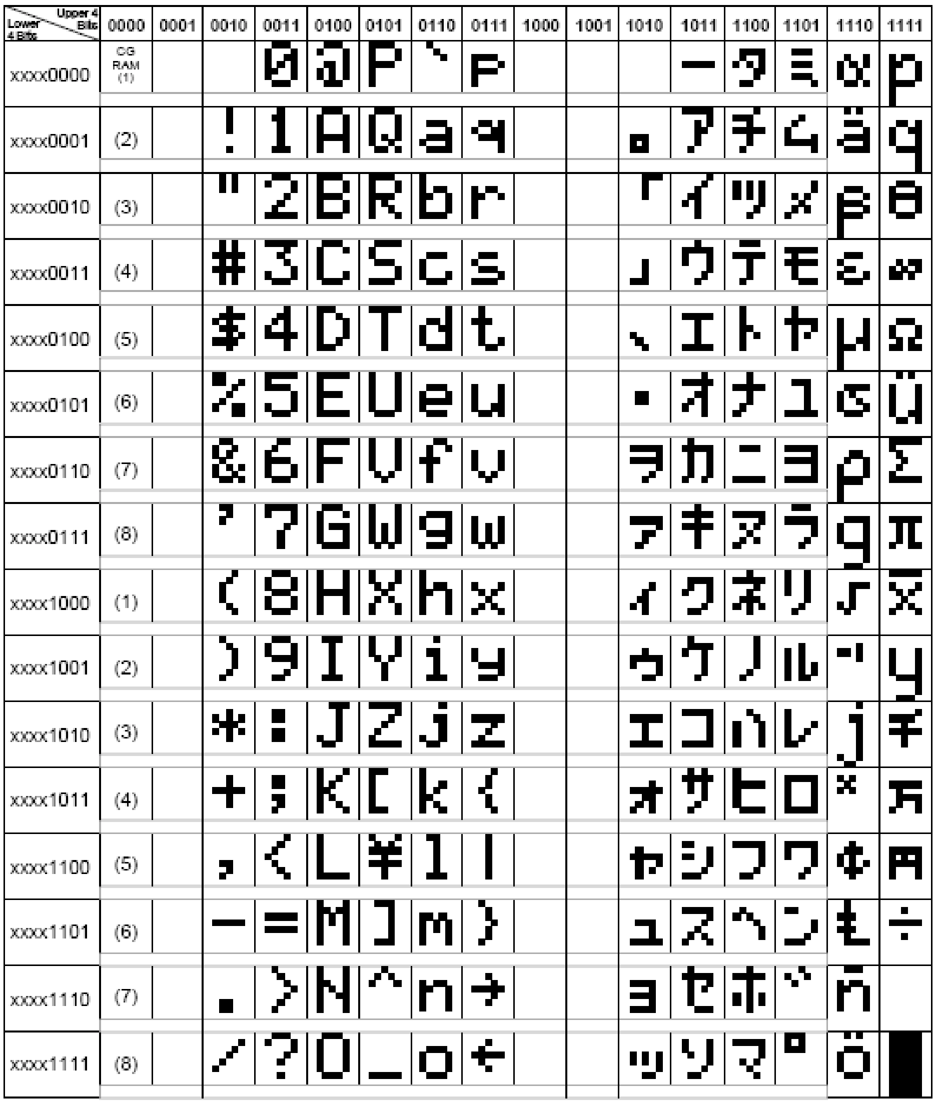
In the initialization procedure we begin setting the data and the control lines to enable the communication between the XE167 microcontroller and the display module; after that we send the "function set" command to the controller specifying an 8 bit data interface, a 5 x 8 character size and 2 lines of text (the HD44780 controller handles the other two text lines as the continuation of the first two lines).

The next step is to power off the display and to disable the visualization of the cursor, then there is the selection of the cursor movement direction: we enable its increments to the right and we disable the shift of the entire display.

After that we clear the entire display and then we power it on: from this point it is ready for the writing process.

The "dispON" and "dispOFF" procedures are trivial: you can use them to hide to the user the writing process and the "clear" and "home" procedures are simple too, they refer to the commands just seen before.

The "writing" procedures lead to the "write data" primitive and they use the built-in character map that is shown in figure 14; this map is contained in the HD44780 ROM memory.



Upper 4 Bits / Lower 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	p				ー	タ	ミ	α	p
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	メ	β	θ
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t			、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	ユ	σ	ü
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		'	7	G	W	g	w			ァ	キ	ヌ	ラ	g	π
xxxx1000	(1)		(	8	H	X	h	x			ィ	ク	ネ	リ	√	x̄
xxxx1001	(2)		)	9	I	Y	i	y			ゥ	ケ	ノ	ル	⁻¹	y
xxxx1010	(3)		*	:	J	Z	j	z			ェ	コ	ハ	レ	j	千
xxxx1011	(4)		+	;	K	[	k	{			ォ	サ	ヒ	ロ	ˣ	万
xxxx1100	(5)		,	<	L	¥	l	|			ャ	シ	フ	ワ	¢	円
xxxx1101	(6)		-	=	M	]	m	}			ュ	ス	ヘ	ン	£	÷
xxxx1110	(7)		.	>	N	^	n	→			ョ	セ	ホ	゛	ñ	
xxxx1111	(8)		/	?	O	_	o	←			ッ	ソ	マ	゜	ö	█

Figure 14 The HD44780 internal character table

Since the same controller is used in combination with displays of different shapes and sizes, before you start to write something on the display you need to know where the DDRAM is mapped in the controller's addressing space; for the TM204 display module the four text lines are mapped in the DDRAM at:

- Line 1: from 00h to 13h;
- Line 2: from 40h to 53h;
- Line 3: from 14h to 27h;
- Line 4: from 54h to 67h.

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53
14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27
54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	60	61	62	63	64	65	66	67

Figure 15 DDRAM address map

As described before, the HD44780 display controller allows you to create up to 8 custom characters: you can do this by writing the character data to this CGRAM address range: 00h – 3Fh. Every custom character will be defined sending eight bytes of data to the controller but, for every byte, only the first five bits will be considered given that the character pixel matrix is composed by a 5 x 8 pixels. Summarizing that:

- Custom character 0: from 00h to 07h;
- Custom character 1: from 08h to 0Fh;
- Custom character 2: from 10h to 17h;
- Custom character 3: from 18h to 1Fh;
- Custom character 4: from 20h to 27h;
- Custom character 5: from 28h to 2Fh;
- Custom character 6: from 30h to 37h;
- Custom character 7: from 38h to 3Fh;

Following these rules and adding the specific command code to the local address we get the global address value referred to the controller's address space so the following procedures will be clear to understand.

```
void LCDgoToPos(ubyte line, ubyte pchar) //connected display is 4 x 20
{
switch (line)                           // use set DDRAM address command
   {                                   // lines order: 1 3 2 4
   case 1:
      LCDwriteCtrl(0x80+pchar-1);
         break;
   case 2:
      LCDwriteCtrl(0xc0+pchar-1);
         break;
   case 3:
      LCDwriteCtrl(0x94+pchar-1);
         break;
   case 4:
      LCDwriteCtrl(0xd4+pchar-1);
         break;
   default: break;
   }
}

void LCDdefineChar(ubyte charNo, ubyte data[8])
{
ubyte i;

LCDwriteCtrl(0x40 + (charNo << 3));     // set CGRAM address command
for(i = 0 ; i < 8 ; i++) LCDwriteData(data[i]);
}

void LCDloadArrowChars(void)
{
const ubyte upArrow[8]    = {0x04,0x0e,0x15,0x04,0x04,0x04,0x04,0x04};
const ubyte downArrow[8]  = {0x04,0x04,0x04,0x04,0x04,0x15,0x0e,0x04};
const ubyte rightArrow[8] = {0x00,0x04,0x02,0x1f,0x02,0x04,0x00,0x00};
const ubyte leftArrow[8]  = {0x00,0x04,0x08,0x1f,0x08,0x04,0x00,0x00};
const ubyte upRarrow[8]   = {0x00,0x07,0x03,0x05,0x08,0x10,0x00,0x00};
const ubyte upLarrow[8]   = {0x00,0x1c,0x18,0x14,0x02,0x01,0x00,0x00};
const ubyte downLarrow[8] = {0x00,0x00,0x01,0x02,0x14,0x18,0x1c,0x00};
const ubyte downRarrow[8] = {0x00,0x00,0x10,0x08,0x05,0x03,0x07,0x00};

LCDdefineChar(0,upArrow);
LCDdefineChar(1,downArrow);
LCDdefineChar(2,rightArrow);
LCDdefineChar(3,leftArrow);
LCDdefineChar(4,upRarrow);
LCDdefineChar(5,upLarrow);
LCDdefineChar(6,downLarrow);
LCDdefineChar(7,downRarrow);
}
```

The last procedure is an example of how to define some custom characters following the rules just described: in this case we have created a set of arrows that point in eight different directions.

Now we have all the elements to write some text on the display; in the next section we will try something of higher level using our customized arrows.

3.3 High level application examples

Now you know how to write your strings on a display but you may want to do this in a more impressive way; so you could try this blinking string procedure.

```
LCDinitialize();
LCDclear();
LCDgoToPos(2,2);
LCDwriteString("Infineon Tech. AG");
while(1)
   {
   LCDdispON();
   wait100ms();
   wait100ms();
   LCDdispOFF();
   wait100ms();
   wait100ms();
   }
```

But you can match static and blinking text with a bit of fantasy.

If you want to make your text as an advertising banner you can try this simple method for moving strings.

```
LCDinitialize();
LCDclear();
while(1)
   {
   wait100ms();
   wait100ms();
   LCDgoToPos(dispYold,dispXold);
   LCDwriteString("                 ");

   switch (direction)
        {
        case RIGHT:
        {
        dispX++;
        if (dispX == 4) direction = DOWN;
        break;
        }
        case DOWN:
        {
        dispY++;
        if (dispY == 4) direction = LEFT;
        break;
        }
        case LEFT:
        {
        dispX--;
        if (dispX == 1) direction = UP;
        break;
        }
        case UP:
        {
        dispY--;
        if (dispY == 1) direction = RIGHT;
        break;
        }
        }
   LCDgoToPos(dispY,dispX);
   LCDwriteString("Infineon Tech. AG");
   dispXold = dispX;
   dispYold = dispY;
   }
```

Of course you can find another way to do this, for example you can save some CPU time shifting the entire screen instead of deleting and re-writing the same string.

Customized characters are very useful to build some simple graphics: look at this flag built with seven different characters.

```
void LCDloadFlagChars(void)
{
const ubyte a1[8] = {0x00,0x00,0x0d,0x0e,0x0c,0x0c,0x0c,0x0c};
const ubyte a2[8] = {0x00,0x00,0x13,0x0c,0x08,0x08,0x08,0x08};
const ubyte a3[8] = {0x00,0x00,0x06,0x1a,0x12,0x12,0x12,0x12};
const ubyte b1[8] = {0x0c,0x0c,0x0c,0x0c,0x0c,0x0d,0x0e,0x0c};
const ubyte b2[8] = {0x08,0x08,0x08,0x08,0x08,0x19,0x0e,0x00};
const ubyte b3[8] = {0x12,0x12,0x12,0x12,0x12,0x16,0x18,0x00};
const ubyte c1[8] = {0x0c,0x0c,0x0c,0x0c,0x0c,0x0c,0x0c,0x00};

LCDdefineChar(0,a1);
LCDdefineChar(1,a2);
LCDdefineChar(2,a3);
LCDdefineChar(3,b1);
LCDdefineChar(4,b2);
LCDdefineChar(5,b3);
LCDdefineChar(6,c1);
}


//...main...
LCDinitialize();
LCDclear();
LCDloadFlagChars();
LCDgoToPos(1,8);
LCDwriteChar(0);
LCDwriteChar(1);
LCDwriteChar(2);
LCDgoToPos(2,8);
LCDwriteChar(3);
LCDwriteChar(4);
LCDwriteChar(5);
LCDgoToPos(3,8);
LCDwriteChar(6);
```

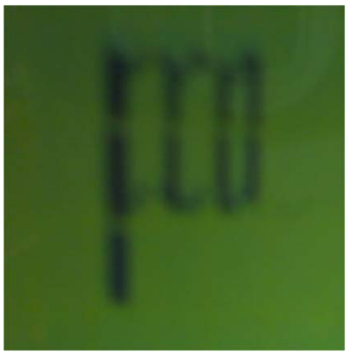


Figure 16 Example of a simple graphic

With our arrows defined before we can create a simple animation with only few lines of code: a rotating arrow.

```
// definitions for customized symbols
#define ARROW_U      0
#define ARROW_D      1
#define ARROW_R      2
#define ARROW_L      3
#define ARROW_UR     4
#define ARROW_UL     5
#define ARROW_DL     6
#define ARROW_DR     7

//…

LCDinitialize();
LCDclear();
LCDloadArrowChars();
while(1)
   {
   LCDgoToPos(2,10);
   LCDwriteChar(ARROW_U);
   wait100ms();
   LCDgoToPos(2,10);
   LCDwriteChar(ARROW_UR);
   wait100ms();
   LCDgoToPos(2,10);
   LCDwriteChar(ARROW_R);
   wait100ms();
   LCDgoToPos(2,10);
   LCDwriteChar(ARROW_DR);
   wait100ms();
   LCDgoToPos(2,10);
   LCDwriteChar(ARROW_D);
   wait100ms();
   LCDgoToPos(2,10);
   LCDwriteChar(ARROW_DL);
   wait100ms();
   LCDgoToPos(2,10);
   LCDwriteChar(ARROW_L);
   wait100ms();
   LCDgoToPos(2,10);
   LCDwriteChar(ARROW_UL);
   wait100ms();
   }
```

You can use your customized characters to build a gauge for any kind of quantities; look at these examples.

```
void LCDloadENGchars(void)
{
const ubyte frame[8]  = {0x1f,0x11,0x11,0x11,0x11,0x11,0x11,0x1f};
const ubyte gauge1[8] = {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1f};
const ubyte gauge2[8] = {0x00,0x00,0x00,0x00,0x00,0x00,0x1f,0x1f};
const ubyte gauge3[8] = {0x00,0x00,0x00,0x00,0x00,0x1f,0x1f,0x1f};
const ubyte gauge4[8] = {0x00,0x00,0x00,0x00,0x1f,0x1f,0x1f,0x1f};
const ubyte gauge5[8] = {0x00,0x00,0x00,0x1f,0x1f,0x1f,0x1f,0x1f};
const ubyte gauge6[8] = {0x00,0x00,0x1f,0x1f,0x1f,0x1f,0x1f,0x1f};
const ubyte gauge7[8] = {0x1f,0x1f,0x1f,0x1f,0x1f,0x1f,0x1f,0x1f};

LCDdefineChar(RPM_FRAME,frame);
LCDdefineChar(GAUGE_ENG_L1,gauge1);
LCDdefineChar(GAUGE_ENG_L2,gauge2);
LCDdefineChar(GAUGE_ENG_L3,gauge3);
LCDdefineChar(GAUGE_ENG_L4,gauge4);
LCDdefineChar(GAUGE_ENG_L5,gauge5);
LCDdefineChar(GAUGE_ENG_L6,gauge6);
LCDdefineChar(GAUGE_ENG_L7,gauge7);
}
```

```
#define DISPLAY_NCHAR 20
#define DISPLAY_NLINE 4

#define RPM_FRAME     0
#define GAUGE_ENG_L1  1
#define GAUGE_ENG_L2  2
#define GAUGE_ENG_L3  3
#define GAUGE_ENG_L4  4
#define GAUGE_ENG_L5  5
#define GAUGE_ENG_L6  6
#define GAUGE_ENG_L7  7

#define CHAR_BLANK    0xfe
#define CHAR_BLACK    0xff

void gaugeExample(void)
{

//...

LCDgoToPos(1,1);
blackChars = (ubyte)(RPM*DISPLAY_NCHAR/maxRPM);    //RPM bar level calculation
for (i = 0 ; i < blackChars ; i++) LCDwriteChar(CHAR_BLACK);
for (i = blackChars ; i < DISPLAY_NCHAR ; i++) LCDwriteChar(RPM_FRAME);
```

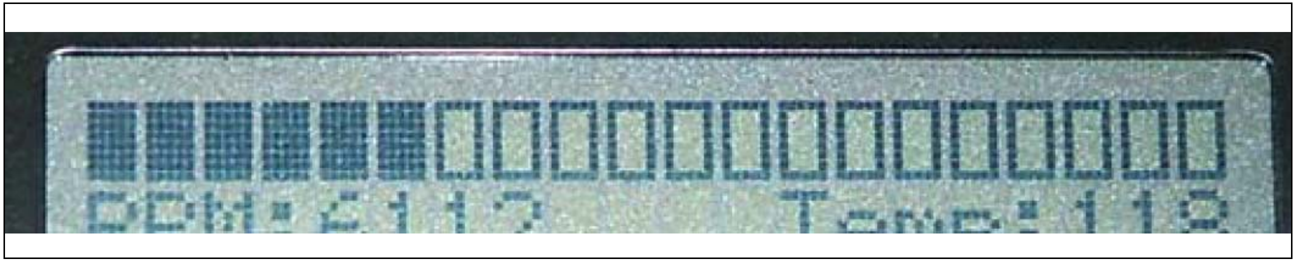


Figure 17 Example of a level indicator 1

```
tempGaugeLevel = (ubyte)((engineTemp * 16)/ maxTemp);
switch (tempGaugeLevel)
   {
   case 0:
         {
         LCDgoToPos(3,19);
         LCDwriteChar(CHAR_BLANK);
         LCDgoToPos(4,19);
         LCDwriteChar(CHAR_BLANK);
         break;
         }
   case 8:
         {
         LCDgoToPos(3,19);
         LCDwriteChar(CHAR_BLANK);
         LCDgoToPos(4,19);
         LCDwriteChar(CHAR_BLACK);
         break;
         }
   case 16:
         {
         LCDgoToPos(3,19);
         LCDwriteChar(CHAR_BLACK);
         LCDgoToPos(4,19);
         LCDwriteChar(CHAR_BLACK);
         break;
         }
   default:
         {
         if (tempGaugeLevel < 8)
            {
            LCDgoToPos(3,19);
            LCDwriteChar(CHAR_BLANK);
            LCDgoToPos(4,19);
            LCDwriteChar(tempGaugeLevel);  //capitalize the character code here
            }
         else
            {
            LCDgoToPos(3,19);
            LCDwriteChar(tempGaugeLevel-8);//capitalize the character code here
            LCDgoToPos(4,19);
            LCDwriteChar(CHAR_BLACK);
            }
         break;
         }
   } // end switch
```

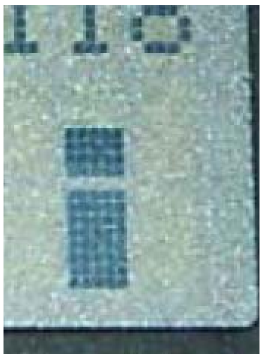


Figure 18 Example of a level indicator 2

4 Conclusions and future improvements

This application note has described how to interface a TM204 display module with an XE167 microcontroller using its parallel ports; this kind of display has a built-in controller that is accessed following specific rules about timings, signals and commands: all these aspects have been clearly explained.

Add a display to your application! It could be necessary if you want to show any kind of size to the user; in this case do it with style, using flashing strings or gauges. Otherwise why don't you add a display to your application however? It's simple to connect and it may help you to debug code, to show the status of a process, to report the statistics of a communication channel and many more, whatever that comes to mind!

To add a semi-graphical display module to your application is easy and cheap; and you remember that it is portable to lower family Infineon's microcontrollers, without any problems.

This application note has shown a possible use of a XE167 microcontroller but this is only a little subset of the capabilities that the Infineon Technologies microcontrollers can deploy. This project could be taken as a little brick to build a more complex application.

Future improvements of this project could be:

- Writing an EBC enhance version;
- Adding the support for displays of different sizes;
- Adding the support for different display controllers.