Digital Systems, Project laboratory EDI 021

Magnus Johansson, e00mj

March 1, 2005

Abstract

Building a complete system with inputs and outputs based on a microprocessor includes several aspects concerning hardware design, noise preventing and low-level programming.

This project is intended to give hands-on experience in these areas. It embraces all phases in studying of data sheets, constructing a hardware prototype and implementing special designed software for the prototype. A main goal has been to show how similar embedded constructions work. By using several different peripheral units and features, a wider knowledge of such systems and their behaviour is shown. A graphical LCD is used as an output unit. Controlling such a device demonstrates software issues related to low-level programming.

With the help of extensive analysis of different problems a working prototype has been built. The functionality is demonstrated by a simple variant of the Pong game

Contents

1 Introduction

The purpose of this project is to introduce the students to hardware design and implementations based on a microprocessor or a micro-controller. The task should illustrate an industrial development process of a prototype. This includes constructing, building, testing the construction and producing a written documentation of the work, design and result.

The constructing involves analysis of the specification, choosing appropriate components and identifying necessary signals and logic from data sheets. The physical design and layout of the prototype must also be taken into consideration. Furthermore, a program for the microprocessor should be implemented, verified and uploaded to the model.

The goal is a fully functional and working prototype that follows the specification. By reaching this, the students should have good insight in industrial development of microprocessor-based designs.

1.1 Problem description

The prototype construction that has been chosen to be built is any type of application using buttons as input and a graphic LCD as output. The construction should be based on a Motorola MC68008 microprocessor and have the ability to save and retain data after power loss. The use of hardware interrupts should be demonstrated.

The minimum software that should be implemented is low-level functions for the hardware. Any other application that is implemented should make the most use of these for accessing and using the peripheral units connected to the microprocessor.

A simple program that demonstrates the prototype's capability and makes use of the low-level functions should also be implemented. In case there would be enough time, a simple game has been proposed as a suitable program to implement.

2 Problem analysis

To find out and decide what hardware should be used, it is necessary to analyse the problem description. The hardware chosen must fulfil the least requirements given in the specification.

2.1 The microprocessor

The microprocessor to be used, the Motorola MC68008, is already given in the problem description. The certain model used in this project is the 48-pin variant. It has the possibility to address a total of 1 Mb and has a eight bits wide data-bus. The processor can be instructed to run in an asynchronous or synchronous mode during run-time.

All peripheral units making use of the data-bus must support tri-state. This means that they besides giving high or low logical level must be able to emulate a high-impedance state. In the high-impedance state, the outputs neither load nor drive the bus lines significantly. This is necessary to avoid any signal collisions, which will give an undefined logical level.

- MC68008 Microprocessor [1]

2.2 System clock

The microprocessor has no internal system clock, which means an external one has to be used. It is not easy to construct an external clock based on a crystal oscillator. Doing this might introduce some problems. To simplify the prototype construction and avoid unnecessary problems, a oscillator chip with a well defined square wave is used. It can easily be configured to run at 8.0000 MHz, which is a reasonable level for the Motorola processor.

- EXO3 16.0000 MHz oscillator [2]

2.3 Memory modules

To make any use of the microprocessor some memory modules must be used. There is a total need of three different types of memory in this construction. One is used for storing the program code for controlling the processor. This memory must be able to retain the data after every power loss, but has no need to be written to in the final construction. This type of memory is called PROM. The second memory needed is an SRAM module. The program uses this to write and read data during run-time. The last memory module used is of the type EEPROM. It can be written to and read from during run-time and will remember the information after a power loss. This module will fulfil the given requirement of that the construction must be able to save and retain data after a power loss.

Since the program to be implemented is at the most a simple game application, no large memory modules are needed. The CPU data-bus is eight bits wide and the memory modules should not be more than that. With these specifications given, the following modules were chosen.

- 27C64, 8kb, 8-bit parallel EPROM [3]

- 6264, 8 kb, 8-bit parallel SRAM [4]
- 28C64, 8kb, 8-bit parallel EEPROM [5]

2.4 Input buttons

Buttons used in a digital architecture should give either high or low logic level. This is easily implemented using ordinary push buttons and pull-up resistors between the ground and load current at the circuit board. Since the buttons should be connected to the data-bus they must also support tri-state. This can be achieved by passing the signals through a tri-state latch.

- 74HC373, 8-bit tri-state latch [6]

2.5 Graphic LCD

A graphic LCD that was available for use during this project consisted of one 128 x 64 dots display driven by two graphic LCD-driver chips controlling half the display each. These require that the signals given at the data-bus are synchronised with the system clock. To realise synchronous mode with the 48-pin variant of the MC68008 microprocessor, two J-K flip-flop circuits are needed.

- BT12064B Graphic LCD [7]
- KS108B Graphic LCD-driver [8]
- 74HC73 Dual J-K Flip-Flop Reset [9]

2.6 Interrupts

A given requirement in the prototype specification is that hardware interrupts should demonstrated. This means that certain interrupt pins at the microprocessor should be issued to change the internal state of the CPU and run specific interrupt routines in the program.

To make a simple realisation of this a 14-bit counter is used. It should increase its value with the help of the system clock and generate an interrupt signal when it has reached a certain value. Such a construction will guarantee that the code written in the corresponding interrupt routine will be executed on a strict regular basis.

- 74HC4020 14-bit counter [10]

2.7 Programmable logic

To take control of all signals, select which unit that should have access to the data-bus and generate asynchronous handshake signals, logic is needed. This can be realised with ordinary NAND-gate logic, but when several input signals must be used to generate multiple outputs it is a hard work. A quicker, cheaper and much more convenient way is to implement it with PAL circuits (Programmable Array Logic). These can emulate complex logic from a specific programming language that is compiled and uploaded to the chip. Once it has been programmed, it will remember its logic even after power loss.

- PALCE22V10 programmable logic [11]

3 Hardware design and logic

Besides choosing what components that is suitable to use, it is also necessary to come up with a design solution that will make all the parts cooperate as they are supposed to. This means that all signals in the prototype must be identified and examined how they should be used.

To reduce possible errors caused by noise, there are several capacitors used to stabilise the power supply to most chips. Many ground points are connected with short wires to construct a type of ground plane for increasing the disturbance tolerance from the surrounding environment.

Another potential source to errors is input pins that are not connected but still used. These are drawn either to ground or power supply to assign them a correct logical level. Otherwise, there is a risk that they may oscillate between the logical levels and cause strange and unpredicted behaviour.

3.1 The addressing

To be able to choose which peripheral unit that should be used, they all must be assigned a unique address. When a program executed in the microprocessor calls a certain address, the hardware should automatically set up the corresponding chip to use the data-bus. This is done by interpreting the address given on the address-pins of the microprocessor in a PAL chip. Once the correct chip has been identified by the logic, a Chip Select signal is sent and the chip leaves the high-impedance state and enters normal logical state at the data-bus.

In this prototype, the MC68008 has the ability to address up to 1 MB with 20 pins. Since the memory modules used are much smaller, they need 13 bits only to address all their data. The button latch and the LCD need five unique addresses in total together. This means that there are seven bits on the CPU address-bus left to use for simple addressing logic.

Chip	*A15*	*A14*	*A13*	*A12-A0*	*Address*
EPROM	0	0	0	x	0x0000 - 0x1FFF
SRAM	0	0	1	x	0x2000 - 0x3FFF
EEPROM	0	1	0	x	0x4000 - 0x5FFF
Button latch	0	1	1	x	0x6000
LCD1	1	0	0	x	0x8000 - 0x8001
LCD2	1	0	1	x	0xA000 - 0xA001

When an address in the table is called, it will be possible to determine which chip that should be selected by considering the three address bits A13-A15. All the memory modules need the twelve first address bits as input to choose correct data. The button latch has no more data than the actual configuration of pressed buttons. Hence, it does not need any input. The LCD uses two addresses. One is used when the data-bus on the LCD should be used for passing instructions and the other one is used when actual data is sent or received. This is realised by using the first address bit, A0, as a signal to the LCD for indicating instruction or data transfer.

The Motorola 68008 microprocessor is designed in such a way that it assumes that the very first part of the addressing space is dedicated for special purposes.

There are well defined addresses where initial value of the Program Counter and information about interrupts are placed. This means that these addresses must be dedicated for the same purpose in the hardware design as well. It is only the EPROM that can contain such information at system start-up. Hence, the EPROM must be in this dedicated addressing space. The other units can be placed arbitrary within the addressing space.

3.2 Asynchronous and synchronous mode

As earlier mentioned, the MC68008 can run in either asynchronous or synchronous mode. The asynchronous cooperation with the peripheral units is accomplished through a set of handshake signals. These are named Address Strobe, Read/Write, Data Strobe and Data Transfer Acknowledgement (AS, R/W, DS, DTACK). How these are used is described in the data sheet for the microprocessor [1]. When synchronous mode is used, the data will appear on the data-bus synchronised with the processor E signal. The LCD unit is the only unit requiring this in this construction.

The 48-pin variant of the MC68008, which is used in this prototype, is unfortunately missing a certain Valid Memory Address output signal which indicates that there is a valid address on the address-bus and that the processor is synchronised with the E signal. This signal can, however, be produced by an external circuit. Together with the CPU input signal Valid Peripheral Address and the Enable signal (VPA, E), the VMA will be sufficient to make the microprocessor to work in synchronous mode.

3.3 Interrupts

The interrupt routines are called based on the interrupt level given on certain input signals to the processor (IPL2/0, IPL1). The Function Codes output signals (FC0-FC2) will indicate if the interrupt was accepted.

In this construction, automated vectoring interrupts are used, which is indicated by using the VPA signal to the processor.

3.4 Generating the signals

Most control signals that are sent to the different peripheral units must be generated by logic from a set of input signals and certain conditions. Chip Select, Read/Write, handshake and synchronous/asynchronous signals are all generated in the PAL-circuits.

The logic that selects the correct chip based on the addressing must consider and generate appropriate handshake signals. When the LCD is issued, the synchronised mode must be realised through logic with signals.

The DTACK signal, which tells the CPU that there is valid data on the data-bus, is generated at the very same time as the chip select to the memory modules. This is possible since the memories are significantlt much faster than the processor, which is based on the system clock. The memories can easily deliver the demanded data before the processor will read from its data-bus.

To implement all necessary logic, two PALCE22V10 must be used to get enough output pins. Another device with more pins could have replaced them both.

See appendix A to get the full logic behind the generated signals. Appendix B shows the circuit diagram of the construction.

4 Software design

When software is built for an embedded design, there are some aspects that must be taken into consideration. When the microprocessor starts it takes for granted that certain memory addresses contains the information necessary to start executing the code instructions properly [1]. To be able to place instructions and information on an exact given position, assembly instructions must be used. Assembly instructions must also be used when full accesses to internal register values are needed.

Variable initialisation on declaration is another part that must be discussed while developing software for an embedded system. If a global variable in an application written in C is given an initial value, it will not be placed in the SRAM. The application C code is compiled on a developer's workstation to produce the machine code that later should be downloaded to the PROM. At this point, the program can only create a virtual image of what should be written to the SRAM. This virtual image will be lost if it is not explicitly taken care of. This can be done by dedicate a certain area of the PROM to initial SRAM values. This area should be copied into the SRAM at start-up of the system. Another solution is to restructure the C code. By initiate global variables at run-time, they will be placed in the correct SRAM. If neither of these solutions is applied, the SRAM area will not be initialised and give an erroneous value on access.

4.1 Necessary assembly code

In this construction is assembly code used to set initial values of important registers, such as the Program Counter, the Stack Pointer, the Frame Pointer and the Status Register. It is important to give these proper values. If not, the correct program instructions will not be executed and the memory usage will not function.

Assembly code must also be used to enable interrupts since this is done by changing a register value. Interrupts must not be enabled until all initialisation code is done. A good point to enable them is in the C main function.

When an interrupt occur, any ongoing execution will be interrupted and the corresponding interrupt routine will be called. To ensure that the interrupted execution will not fail when returned, some register values must be stored and restored. This is achieved with the help of assembly code.

4.2 Low-level drivers

To make the application programming for the construction easier and straightforward, it is good to implement functions that handle the hardware related issues. These functions should take care of addressing, timing and other special capabilities and limitations that should not be seen or used in the application code.

The memory modules and the buttons can easily be accessed in ordinary manner through a simple address pointer. These system specific addresses can be hidden in macros to remove all address literals from the application code.

The LCD is, on the other hand, a bit more complicated to control. It is controlled by a certain interface defined in the LCD-driver data sheet [8].

Certain bit-patterns are translated as instructions in the LCD-driver circuit and should definitely be taken care of by dedicated functions, or low-level drivers.

4.3 The LCD controlling functions

To help the control of the LCD some basic functions were implemented. Besides the two functions **void lcdOn()** and **void lcdOff()**, which turns on and off the display, two different functions for drawing were considered as a must: **void lcdClear()** and **void lcdPlacePixel(unsigned int x, unsigned int y, unsigned int value)**.

lcdClear clears the display from all lit dots. **lcdPlacePixel** turns on or off a certain dot in the display. This function invokes several instructions that must be sent to and read from the LCD-driver to realise the task. Since the display data-bus is eight bits wide (i.e. eight display dots wide), the old information at the position must be read and the given bit changed before it is written to the display again. This function demonstrates well how a low-level driver hides the hardware related issues and gives a simple interface to the application programmer.

Even though these functions should be enough to use the LCD, they are not the optimal set. If more than one display dot should be altered at the same time, it could be made more effectively since the data-bus is eight bits wide. If, for instance, text should be displayed it means that several dots should be changed after a pre-defined set of images, i.e., a font. By knowing that several of the eight display dots should be changed in the same position, several instructions could be removed in an optimised function. Low-level drivers should not be unnecessarily slow if it can be avoided.

This project has a main goal of showing different aspects and problems related to constructing a system based on a microprocessor. This does not include implementing several various and optimised low-level drivers, but showing and discussing the technique. The **lcdPlacePixel** does that. See appendix C for the code listing.

5 Conclusions

Developing a system based on a microprocessor is not trivial. There are many aspects that must be considered from the very beginning. It is not possible to construct a full prototype without knowing exactly how all the different components behave and interact with each other. One must study the data sheets, build up and verify the prototype one part at the time.

In this project, studying data sheets has been an essential part of the constructing. Several sketches has been drawn and rejected before the final solution. When the understanding of the different components comes with the studying, the realisation of the construct is by far easier. A project that seemed to be fairly advanced and time-consuming turned out to be quite simple to realise and without any serious problems.

Using a well-defined oscillator as a clock, noise reducing capacitors and many ground pins connected as a plane, might very well have saved a lot of time. There have not been any problems at all related to such disturbances.

Developing the software for an embedded system gives quite some new types of challenges. Creating low-level drivers is a bit complicated from the beginning and without any possibility of using feedback in shape of text at a screen it does not get easier. The debugging process had to be made with the help of the development kit, stepping through the machine code and keeping track of register values. This is a completely new level of debugging compared to software development on a PC.

However, the goal in the problem description has been met. The prototype is working at its present state and the input/output functionality is demonstrated with a simple Pong-like game.

References

[1] MC68008 Microprocessor
http://www.it.lth.se/datablad/Processors/68000UM.pdf

[2] EXO3 16.0000 MHz oscillator
http://www.elfa.se/pdf/74/07454002.pdf

[3] 27C64, 8kb, 8-bit parallel EPROM
http://www.it.lth.se/datablad/Memory/eprom/27c64.pdf

[4] 6264, 8 kb, 8-bit parallel SRAM
http://www.it.lth.se/datablad/Memory/sram/HM6264.pdf

[5] 28C64, 8kb, 8-bit parallel EEPROM
http://www.it.lth.se/datablad/Memory/eeprom/at28C64B.pdf

[6] 74HC373, 8-bit tri-state latch
http://www.it.lth.se/datablad/Logik/74HC/74HC373.pdf

[7] BT12064B Graphic LCD
http://www.it.lth.se/datablad/display/Batron128x64.pdf

[8] KS108B Graphic LCD-driver
http://www.it.lth.se/datablad/display/ks0108b.pdf

[9] 74HC73 Dual J-K Flip-Flop Reset
http://www.it.lth.se/datablad/Logik/74HC/74HC73.pdf

[10] 74HC4020 14-bit counter
http://www.it.lth.se/datablad/Logik/74HC/74HC4020.pdf

[11] PALCE22V10 programmable logic
http://www.it.lth.se/datablad/logik/Programmable/Palce22v10.pdf

A PAL logic

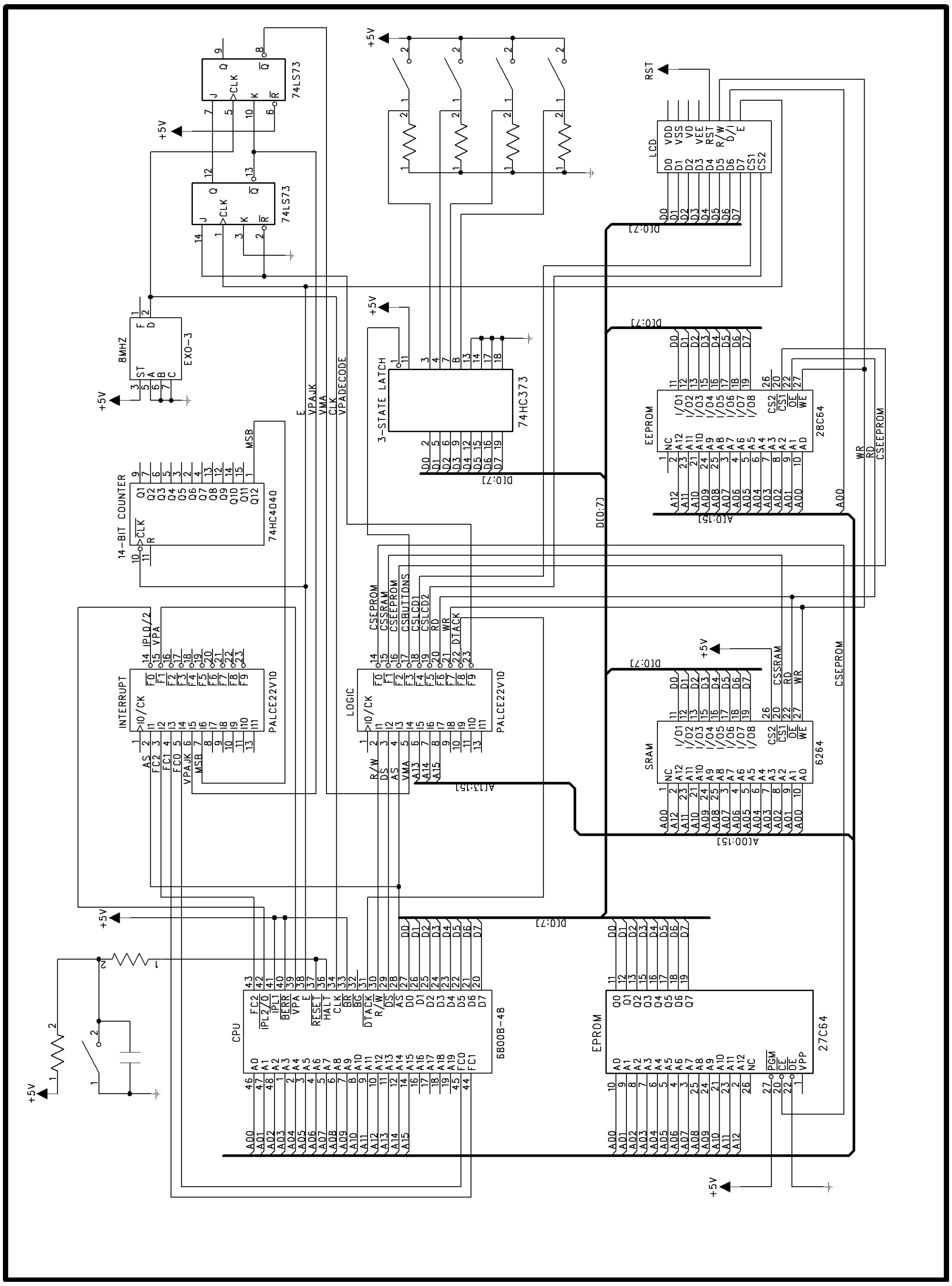
Short PAL logic language description:

Logic	*Symbol*
not	/
and	*
or	+

```
Title     Digital Project
Pattern   Memory and I/O
Author    Magnus Johansson
Date      7 February 2005

device 22V10
RW        2   'Read/Write signal from CPU
DS        3   'Data Strobe from CPU
AS        4   'Address Strobe from CPU
VMA       5   'VMA signal from emulating J/K flip-flop
A13       6   'Address bit 13 from CPU
A14       7   'Address bit 14 from CPU
A15       8   'Address bit 15 from CPU
GND       12
CSEPROM   14  'Chip Select to EPROM memory
CSSRAM    15  'Chip Select to SRAM memory
CSEEPROM  16  'Chip Select to EEPROM memory
CSBUTTONS 17  'Chip Select to button latch
CSLCD1    18  'Chip Select to first LCD segment
CSLCD2    19  'Chip Select to second LCD segment
RD        20  'Read signal to memory and I/O
WR        21  'Write signal to memory and I/O
DTACK     22  'Data Transfer Acknowledge to CPU
VPADECODE 23  'VPA Decode signal to VMA emulating J/K flip-flop
VCC       24

start
CSEPROM /= /AS * /A15 * /A14 * /A13;
CSSRAM /= /AS * /A15 * /A14 * A13;
CSEEPROM /= /AS * /A15 * A14 * /A13;
CSBUTTONS /= /AS * /A15 * A14 * A13;
CSLCD1 /= /VMA * /A13; 'The VMA signal contains the VPADECODE attributes
CSLCD2 /= /VMA * A13;
RD /= /DS * RW;
WR /= /DS * /RW;
DTACK /= /CSEPROM + /CSSRAM + /CSEEPROM + /CSBUTTONS;
VPADECODE = /AS * A15 * /A14;
end
```

```
Title     Digital Project
Pattern   Interrupt
Author    Magnus Johansson
Date      7 February 2005

device 22V10
AS        2   'Address Strobe from CPU
FC2       3   'Function Code signal 2 from CPU
FC1       4   'Function Code signal 1 from CPU
FC0       5   'Function Code signal 0 from CPU
VPAJK     6   'Generated VPA signal from VMA emulating J/K flip-flop
MSB       7   'Most Significant Bit from counter
GND       12
IPL02     14  'Interrupt input signal IPL0/IPL2 to CPU
VPA       15  'VPA signal to CPU
VCC       24

start
IPL02 /= MSB;
VPA /= /VPAJK + /AS * FC2 * FC1 * FC0;
End
```

B Circuit diagram



C lcdPlacePixel

```
unsigned short int* lcd1Instr;
unsigned short int* lcd1Data;
unsigned short int* lcd2Instr;
unsigned short int* lcd2Data;

/* Use macro rather than function calls for these simple functions.
   This will slightly increase the speed of the low-level drivers
   thanks to the inlining result. */

#define LCDBUSYWAIT(x) while((*x & 0x80) == 0x80);
#define INITADDRESSES lcd1Instr = (unsigned short int*) 0x8000;\
                      lcd1Data = (unsigned short int*) 0x8001;\
                      lcd2Instr = (unsigned short int*) 0xa000;\
                      lcd2Data = (unsigned short int*) 0xa001;

void lcdPlacePixel(unsigned short int x, unsigned short int y,
                   unsigned short int value)
{
  unsigned short int* lcdInstr = lcd1Instr;
  unsigned short int* lcdData = lcd1Data;
  unsigned char page, bit, pattern, newPattern;

  INITADDRESSES

  if(value != 0){
    value = 1;
  }
  x %= 128;
  if(x > 63){
    lcdInstr = lcd2Instr;
    lcdData = lcd2Data;
    x -= 64;
  }
  y %= 64;
  page = y / 8;
  bit = y - page * 8;

  /* choose correct page and address */
  *lcdInstr = 0xb8 + page;
  LCDBUSYWAIT(lcdInstr)
  *lcdInstr = 0x40 + x;

  /* read current pattern. needs dummy read */
  LCDBUSYWAIT(lcdInstr)
  pattern = *lcdData;
  LCDBUSYWAIT(lcdInstr)
  pattern = *lcdData;

  /* the read instruction ticks the Y address as well. reset for writing */
  LCDBUSYWAIT(lcdInstr)
  *lcdInstr = 0x40 + x;

  /* create a new pattern with the bit set to the new value */
  bit = 0x1 << bit;
  newPattern = (pattern & (~bit)) + bit * value;

    LCDBUSYWAIT(lcdInstr)
  *lcdData = newPattern; /* change the bit */
}
```