ATA I/O Code Snippets

1. Principles

ATA I/O adapters are to be programmed like the I/O ports of microcontrollers. The starting point of programming a particular ATA I/O adapter is its register description and its principles of operation.

Note:
Described here are the most basic ATA I/O adapters, which do not support command execution. They could be only operated on genuine parallel ATA (PATA) host adapters, not on host adapters which are basically protocol converters from serial to parallel ATA.

Machine instructions
ATA I/O programming is based on accessing the ATA register file by means of I/O instructions (Listing 1).

Input:

1. Place the I/O port address in the DX register.

2. Execute an IN AL, DX instruction.

3. Proceed with processing the byte found in the AL register.

Output:

1. Place the byte to be transferred in the AL register.

2. Place the I/O port address in the DX register.

3. Execute an OUT DX, AL instruction.

```
        Input                           Output

MOV DX, port_adrs               MOV AL, byte_value
IN AL, DX                       MOV DX, port_adrs
                                OUT DX, AL
```

Listing 1 Examples of basic input and output operations.

Addressing the ATA port
Usually, ATA ports on the motherboard are assigned legacy I/O addresses (Tables 1 and 2). The addresses of ATA ports on add-in cards are assigned by the plug-and-play software. How to obtain these address values, depends on the particular system environment. The most general approach would be using appropriate API calls (operating system or PnP BIOS). An alternative

method consists in scanning the configuration address space (for example, by calling the PCI BIOS). If the hardware configuration will never change, address assignment could be detected even manually. In case of Windows, simply select Control Panel – System Properties – Hardware – Device Manager – ATA/ATAPI Controllers – Resources. Otherwise, run an appropriate system information or diagnostic program showing the PCI configuration.

IDE/ATA Channel	Device Addresses (Hex)		Interrupt Line	Alternate Interrupt Line
	Comand Block Registers	Control Block Registers		
1. (Primary)	1F0...1F7	3F6	14	-
2. (Secondary)	170...177	376	15	-
3. (Ternary)	1E8...1EF	3EE	11	12 or 9
4. (Quaternary)	168...16F	36	10	

Table 1 Legacy ATA address ranges.

CS		Register Address DA				Register	Legacy PC ATA Ports			
1-	0-	2	1	0	Hex		1	2	3	4
1	0	0	1	0	2	**REG 2** (Sector Count)	1F2	172	1EA	0,17
1	0	0	1	1	3	**REG 3** (LBA Low)	1F3	173	1EB	16B
1	0	1	0	0	4	**REG 4** (LBA Mid)	1F4	174	1EC	16C
1	0	1	0	1	5	**REG 5** (LBA High)	1F5	175	1ED	16D
1	0	1	1	0	6	**DH** (Device/Head)	1F6	176	1EE	16

Table 2 ATA registers applied for general purpose I/O.

ATA I/O and the operating system

Whether or not I/O instructions are permitted in application programs, depends on the operating system. DOS imposes no restrictions at all. In Unix-like systems, an application program can be given rights to execute I/O instructions by appropriate administration. Under Windows, I/O instructions are permitted only in privileged programs. Consequently, accessing I/O ports requires port drivers or specific device drivers.

Implementing the basic I/O access operations

ATA registers are accessible only one byte at a time. The byte transfer can be implemented by inserting machine (assembler) instructions. Most of the compilers provide appropriate functions, however. But beware – there are little differences (Table 3).

TurboC++	PacificC	gcc (Linux)
inportb (adrs) outportb (data, adrs)	inp (adrs) outp (adrs, data)	inb (adrs) outb (data, adrs)

Table 3 Byte I/O functions in some C compilers.

2. Basic code examples

Note:
The following code examples have been prepared using Borland's TurboC++-compiler (now freely available) running under DOS (keep it simple and stupid . . .).

An ATA I/O hardware example
Our first example is the ATA adapter 05a (Fig. 1). It comprises five universal bidirectional 8 bit I/O ports. Under program control, each of the 5 • 8 = 40 I/O lines can be used as an input or as an output. Each I/O port comprises a direction control register (DIR) and a data register (DAT).

Each DIR bit position controls the direction of the corresponding I/O pin (Fig. 2):

- DIR bit = 0: pin configured as an input (driver in high impedance state),
- DIR bit = 1: pin configured as an output (driver active). The potential (low or high) on the pin corresponds to the corresponding bit in the DAT register.

Each of the five ports (A... E) has its own port address in the DH register (Table 4). Within each port, the data register (DAT) as well as the direction control register (DIR) are to be addressed via two of the other ATA registers (Table 5).

Only the potentials (low or high) on the pins can be read back. Each read access to one of the two registers (DIR or DAT, respectively) will deliver the potentials on the pins. This Spartan solution has been chosen in order to keep cost down.

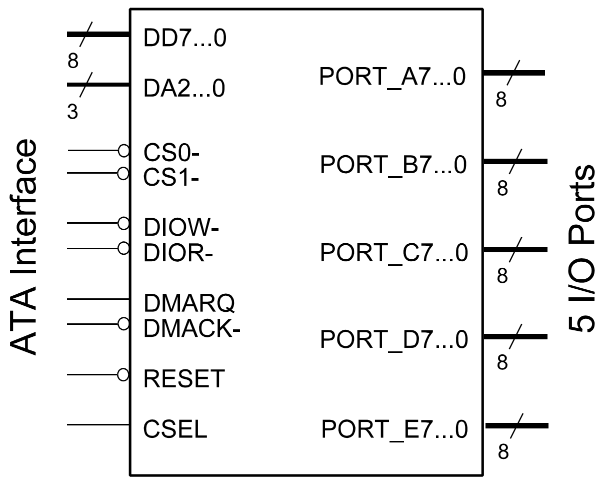

Fig. 1 ATA I/O port adapter 05a comprising five universal I/O ports.

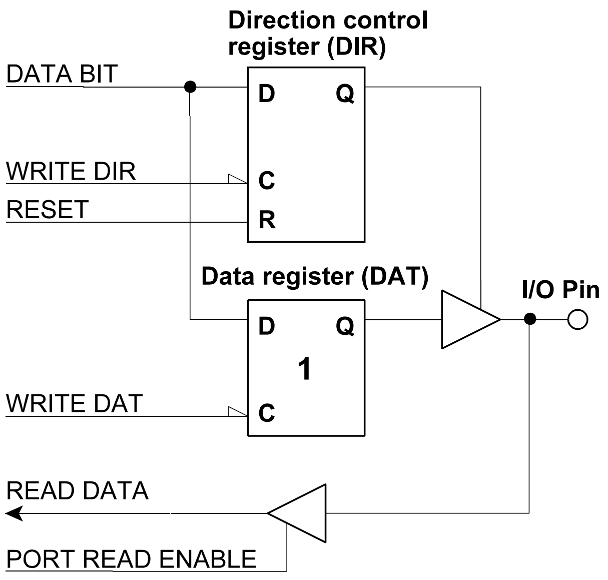

Fig. 2 Principal structure of an I/O bit position.

DH register bits 3... 0	I/O port
1H	Port A
2H	Port B
3H	Port C
4H	Port D
5H	Port E
all other values	no effect

Table 4 Port addressing via DH register.

CS		Register Address DA				Register	Legacy ATA Ports			
1-	0-	2	1	0	Hex		1	2	3	4
1	0	1	0	0	4	Data register (DAT)	1F4	174	1EC	16C
1	0	1	0	1	5	Direction control register (DIR)	1F5	175	1ED	16D
1	0	1	1	0	6	Device selection and port address (DH)	1F6	176	1EE	16

Table 5 Register addressing within a selected I/O port. Read accesses to either DAT or DIR addresses will read back the values on the pins.

Basic declarations

Listing 2 shows the declaration of the physical ATA addresses. In this example, the legacy ATA port 2 is used. In Listing 3, the addressing of the five I/O ports is shown. These values are to be loaded into the DH register. The example ATA I/O adapter is configured as the slave device, hence bit four is always set.

```
//    Physical ATA addresses

//   The ATA I/O adapter is attached to the legacy ATA port 2

#include <stdio.h>,<conio.h>,<dos.h>,<string.h>

#define portadrs      0x176         // DH register
#define dir_reg       0x175         // DIR register of the selected port
#define dat_reg       0x174         // DAT register of the selected port
```

Listing 2 Declaration of physical ATA addresses. This declaration relates to the PC's ATA port to which the adapter is to be attached. Here it is ATA port 2 (example source file ata_phy.c).

```
//      Port addresses for ATA adapter 05a

//      The ATA I/O adapter is the slave device

//      Addresses to be loaded into DH register before accessing the port

#define ioport_a      0x11
#define ioport_b      0x12
#define ioport_c      0x13
#define ioport_d      0x14
#define ioport_e      0x15
```

Listing 3 Declaration of the adapter's port addresses. This declaration pertains to the particular adapter type and to the device configuration at the ATA cable (master or slave). Here the ATA adapter 05a has been configured as slave device (example source file ata_adap.c).

Physical ATA register access

Each I/O register access must be preceded by loading the device selection bit together with the particular port address into the DH register:

1. Load the DH register (Fig. 3).

2. Read or write one of the other registers of the ATA register file (REG 2 to REG 5 according to Table 2).

Listing 4 shows an example.

Notes:

1. The DH register content cannot read back.

2. The effect of a read or write access on an ATA register address depends on the particular adapter (see register description).

3. If the DH register contains already the correct value, it is not necessary to load it again before accessing the other ATA registers.

4. In the time interval between loading the DH register and accessing another ATA register, modification of the DH register content by other programs must be prevented.

   This problem will arise only if (1) together with the ATA I/O adapter a regular drive is attached to the same cable and if (2) the operating system supports preemptive multitasking or if (3) ATA I/O adapters on the same cable are used by different interrupt handlers. Countermeasures are obvious. The most elementary solution: avoid attaching ATA I/O adapters and drives on the same cable. The ultimate solution would be an appropriate kernel driver.

7	6	5	4	3	2	1	0
-	-	-	DEVICE	I/O Port Selection (Port Address)			

Fig. 3 DH register content.

```
//   Physical ATA register access routines for ATA I/O adapter 05a

//   Basic declarations

#include <ata_phy.c>
#include <ata_adap.c>

//                    Borland Turbo C++ physical I/O port access functions used here

//                    If using another compiler, substitute outportb and inputb
                      accordingly

//                    Use physical address decarations only!


// ***************** Load the direction register ******************


void dirout(int adrs, int databyte
{
    outportb (portadrs, adrs); outportb (dir_reg,databyte);
}
```

```
// ***************** Load the data register ******************


void datout (int adrs, int databyte)
{
    outportb (portadrs, adrs); outportb (dat_reg, databyte);
}


// ***************** Read the I/O pins ******************


int datin (int adrs)
{
    outportb (portadrs, adrs); return  inportb (dat_reg);
}
```

Listing 4 Physical ATA register access routines for ATA I/O adapter 05a. This is an extract from the example source file ataio05a.

Supporting modification of register contents

To provide read-back capability for all ATA registers would increase the complexity and hence cost of the adapter hardware. But modification of register contents may be supported by software, however. This support is based on keeping copies of the register contents in RAM (Listing 5). During a write operation (output) the register content will be written into the RAM copy as well as into the hardware register. A read operation (input) addressing a write-only register will fetch the register content out of the RAM copy. A modification operation (for example, setting or clearing particular bits) will (1) modify the RAM copy and (2) write the RAM content into the hardware register.

The RAM copy in the following example (Listing 4) consists of two arrays. The *portregs* array holds the contents. The *portctl* array holds a type designation and an address pointer (Fig. 4).

15 0	7 0
Type designation	Register address

Fig. 4 The structure of one element of the portctl array (see Listing 5). Type designations: 1 = direction control register; 2 = data register; 3 = I/O pins.

```
//   Basic I/O routines for ATA I/O adapter 05a

// *********** Basic declarations

#include <ata_phy_02.c>
#include <ata_adap_05s.c>
```

```
// ************* Logical addresses of the port registers

#define  dir_a       0
#define  dat_a       1
#define  port_a      2
#define  dir_b       3
#define  dat_b       4
#define  port_b      5
#define  dir_c       6
#define  dat_c       7
#define  port_c      8
#define  dir_d       9
#define  dat_d      10
#define  port_d     11
#define  dir_e      12
#define  dat_e      13
#define  port_e     14


// ***************** Global variables ******************************

int portregs [15];      // RAM copies of the port registers

int portctl  [15];      // port access control array

int bitmasks [8];       // masks for single bit access routines


//                      Borland Turbo C++ physical I/O port access functions used here

//                      If using another compiler, substitute outportb and inputb
                        accordingly

//                      Use logical addresses only (according to declarations above)!

// ********************** Single Byte Output ***************************

void out (int adrs, int data)
{

switch ((portctl [adrs] & 0xff00))
{
    case 0x0100:                                    // if direction control register
    portregs [adrs] = data;                         // save copy of new register content
    outportb (portadrs, (portctl [adrs] & 0xff));   // port adrs into DH reg
    outportb (dir_reg,data);                        // data to direction control register
    break;
    case  0x0200:                                  // if data register
    portregs [adrs] = data;                        // save copy of new register content
    outportb (portadrs, (portctl [adrs] & 0xff));  //  port adrs into DH reg
    outportb (dat_reg,data);                       // data to data register
    break;
}

return;
}



// ********************* Single Byte Input **************************

int in (int adrs)

{

if ((portctl [adrs] & 0xff00) == 0x0300)           // only the port will be read in
    {
    outportb (portadrs, (portctl [adrs] & 0xff));  //  port adrs into DH reg
    portregs [adrs] =  inportb (dat_reg);          // update register copy in RAM
    }

return portregs [adrs];                    // read register content out of copy in RAM
}


// ********************* Single Bit Output **************************

//                          Bit modification will be done with the copies in RAM.

//                          Truth values correspond to C conventions.
//                          If data value is zero, bit will be cleared.
//                          If data value is not zero, bit will be set


void bitout (int adrs, int bitpos, int data)

{

switch ((portctl [adrs] & 0xff00))
  {

   case 0x0100:                                              // if direction control register
   if (data == 0) portregs [adrs] = (portregs [adrs] & ~bitmasks [bitpos] &
   0xff);                                                                       // clear bit
   else   portregs [adrs] = (portregs [adrs] | bitmasks [bitpos]);                  // set bit

   outportb (portadrs, (portctl [adrs] & 0xff));                          // port adrs into DH reg
   outportb (dir_reg,portregs [adrs]);                                    // data to data register
   break;

  case  0x0200:                                                           // if data register
   if (data == 0) portregs [adrs] = (portregs [adrs] & ~bitmasks [bitpos] &
   0xff);
   else   portregs [adrs] = (portregs [adrs] | bitmasks [bitpos]);

   outportb (portadrs, (portctl [adrs] & 0xff));                          // port adrs into DH reg
   outportb (dat_reg,portregs [adrs]);                                    // data to data register
   break;
   }

return;
}

 // ********************* Single Bit Input **************************

//                              This routine will return an integer 0 or 1.

int bitin (int adrs, int bitpos)

{

    if ((portctl [adrs] & 0xff00) == 0x300                        // only the port will be read in
    {
    outportb (portadrs, (portctl [adrs] & 0xff));                 //  port adrs into DH reg
    portregs [adrs] =  inportb (dat_reg);                         // data to copy in RAM;
    }

                         // Return bit value out of copy in RAM

    if ((portregs [adrs] & bitmasks [bitpos]) == 0) return 0;
    else return 1;

}



 // ********************* I/O Initialization **************************

//                       To be called at the beginning of the application program.
```

```
void ioinit ()
{
int n;

for (n = 0; n <= 44; n++)
portregs [n] = 0;                                    // clear all copies in RAM

//                      Build the access control array
//                      Low order byte = port address within the ATA I/O adapter
//                      High order byte encodes register type:
//                      1 = direction control, 2 = data, 3 = port

portctl  [0]        = 0x0100+ioport_a;            // port A
portctl  [1]        = 0x0200+ioport_a;
portctl  [2]        = 0x0300+ioport_a;
portctl  [3]        = 0x0100+ioport_b;            // port B
portctl  [4]        = 0x0200+ioport_b;
portctl  [5]        = 0x0300+ioport_b;
portctl  [6]        = 0x0100+ioport_c;            // port C
portctl  [7]        = 0x0200+ioport_c;
portctl  [8]        = 0x0300+ioport_c;
portctl  [9]        = 0x0100+ioport_d;            // port D
portctl [10]        = 0x0200+ioport_d;
portctl [11]        = 0x0300+ioport_d;
portctl [12]        = 0x0100+ioport_e;            // port E
portctl [13]        = 0x0200+ioport_e;
portctl [14]        = 0x0300+ioport_e;

//                  Build the bitmask array

bitmasks[0] = 1;
bitmasks[1] = 2;
bitmasks[2] = 4;
bitmasks[3] = 8;
bitmasks[4] = 16;
bitmasks[5] = 32;
bitmasks[6] = 64;
bitmasks[7] = 128;

}
```

Listing 5 Basic I/O routines for accessing the ATA I/O adapter 05a. These functions emulate five I/O ports supporting three I/O addresses each (DIR, DAT and PORT). The ports behave, for example, like an Atmel AVR Port (with its DDR, PORT and PIN registers). This is an extract from the example source file ataio05a.