Chapter 7

Parallel Input/Output Ports

A microcomputer must be able to communicate with the outside world, i.e., receive information or feedback from the environment, and send out control signals. For example, it should be able to read (or input) the temperature and send out the turn-on/turn-off signals to control an air conditioner. The communication is carried out by the Input/Output (I/O) modules[1] hooked up to the microprocessor, as illustrated in Figure 1 from Chapter 3. There are several I/O modules in a typical microcomputer; this textbook edition only covers parallel I/O ports.

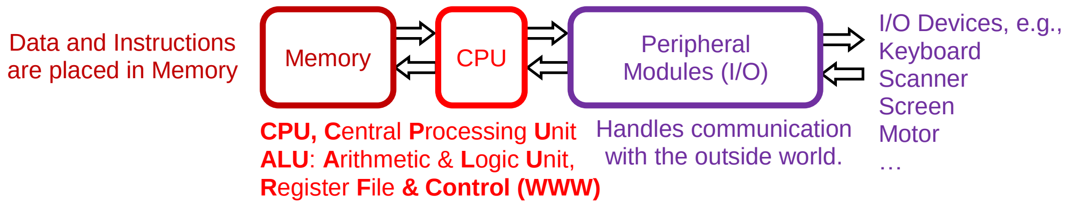

Data and Instructions are placed in Memory
Memory
CPU
Peripheral Modules (I/O)
I/O Devices, e.g., Keyboard Scanner Screen Motor …
CPU, Central Processing Unit
ALU: Arithmetic & Logic Unit,
Register File & Control (WWW)
Handles communication with the outside world.

Figure 1. Simplified block diagram of a microcomputer

The alarm controller example from Chapter 1 is again shown in Figure 2, where the CPU receives two feedback signals, Th and Ph, from a hypothetical chemical plant and sounds an alarm only if both signals are asserted, which happens when the temperature and pressure, respectively, of the plant, rise beyond a threshold. A bare microprocessor has no means to listen to the plant or talk to the alarm. We need to add I/O ports to the microprocessor to facilitate communication, as shown in Figure 2. This improvement is elaborated on in this chapter.

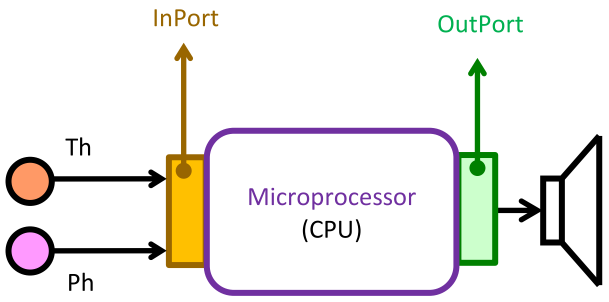


Figure 2. Simplified block diagram of a computer

Figure 3 illustrates a microcontroller's inside and outside worlds; only the memory is shown on the inside; there are an alarm and an LED in the outside world. Let us say the LED turns on with a 1, and the alarm has a built-in oscillator that turns on with a 0. Additionally, the content of memory location 0001 is $8C *before* instruction {movb #$81, $0001} executes, as shown in Figure 3a. Figure 3b shows the content of memory location 0001 *after* the above instruction executes: (**Note**: this instruction uses immediate addressing mode for the source (#$81) and the extended addressing mode for the destination ($0001.)

[1] Some I/O modules, such as timers, help the microcomputer communicate with the outside world, so they may not directly participate in communication.

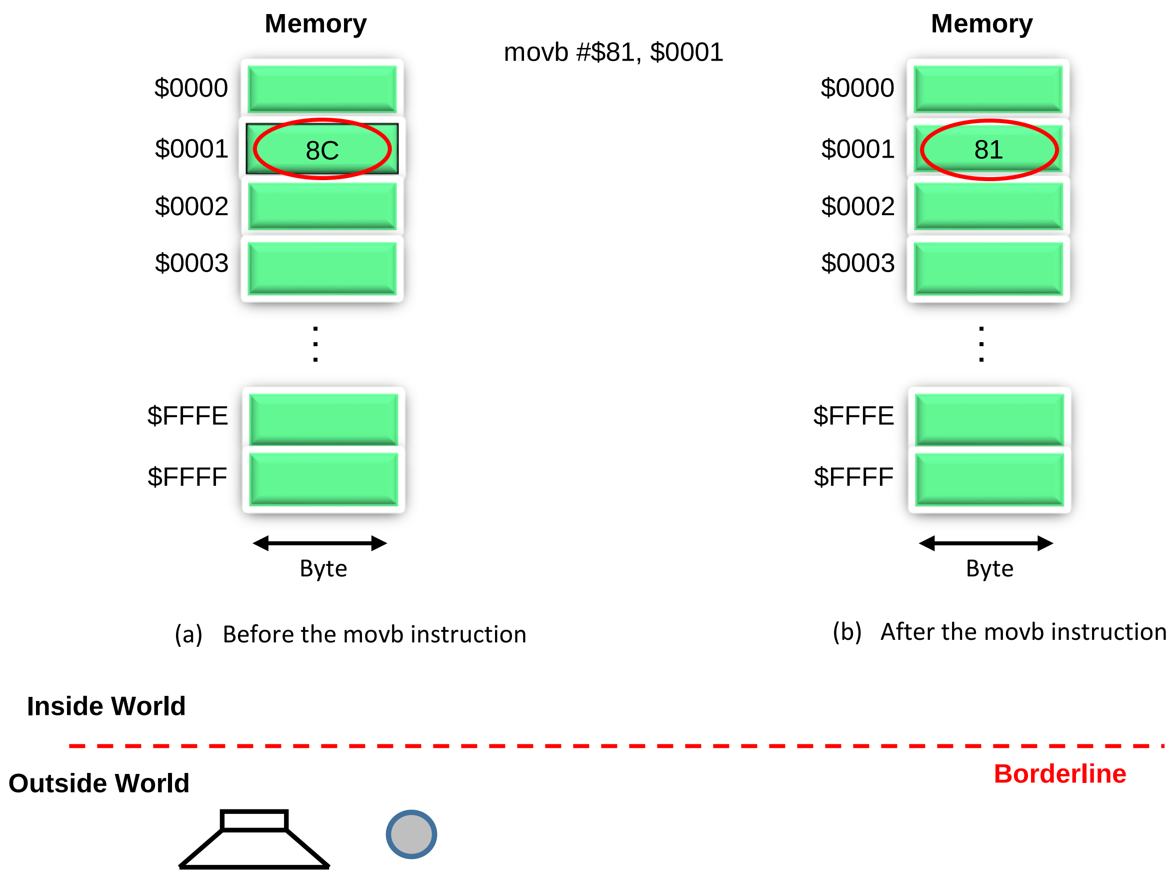


Figure 3. Content of location $0001 (a) before, (b) after {movb #$81, $0001} executes

As you see, the change in the content of location $0001 cannot impact the outside world (the LED or the alarm) as the content of this location is not exposed to the outside world. Now, let us move the memory location at 0001 to the borderline, as shown in Figure 4, and call it the *Data Register* of PORT B or simply PORTB. (Note that PORTB and Register B are two different registers. Also, as you will see shortly, PORTB is not the only register of PORT B.) The 8-bit content of PORTB is exposed to the outside world through the eight *pins* of this port; call them PBs or Bpins, where B is the port name. See Figure 4. Each PB or Bpin gets the index of the associated bit of PORTB, so Bit0 (of PORTB) is connected to Bpin0 (PB0), Bit1 to Bpin1 (PB1), all the way up to Bit7, which is connected to Bpin7 (PB7). We use a similar numbering for the pins of the other ports, comping up.

Back to Figure 4, let us assume that the LED is connected to Bpin0 and the oscillator to Bpin7. We assume that the current content of PORTB is $8C = %1000 1100; therefore, both LED and oscillator are initially turned off.

Let us say the CPU executes instruction {movb #$81, PORTB}, which changes the content of PORTB to %1000 0001, pulling Bpin0 up, hence turning the LED on, as shown in Figure 5. (In assembly language, you do not have to type $0001, the numeric address of PORTB; type PORTB (with no space); the assembler knows what you mean.) So, the microcontroller was able to light up the LED. The alarm remains silent as the MSb of PORTB is still 1. Value %1000 0001 will stay in PORTB unless it is overwritten.

If we execute instruction {movb #$71, PORTB}, value %0111 0001 will be written in PORTB, pulling down Bpin7, hence turning on the alarm as the MSb of the loaded value is 0 while LED remains on because the LSb (of the loaded value) is 1.

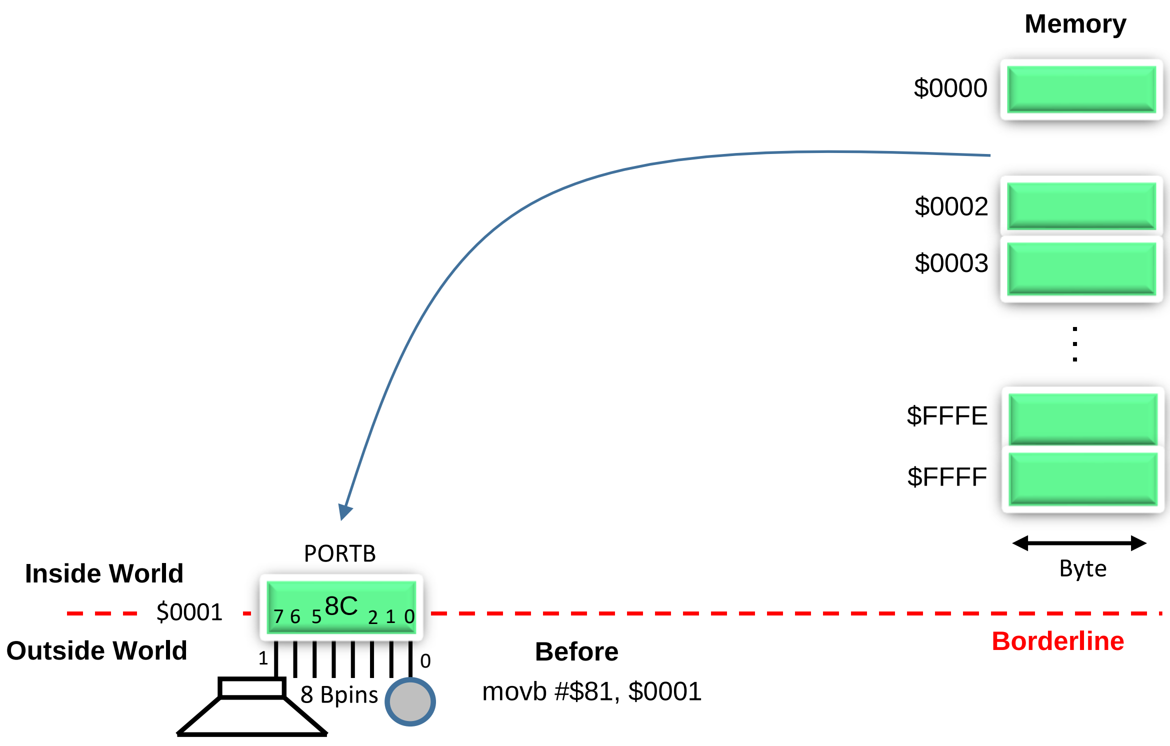


Figure 4. Memory location at $0001 is moved to the borderline. Call it PORTB. Here it drives an LED and an alarm as an example.

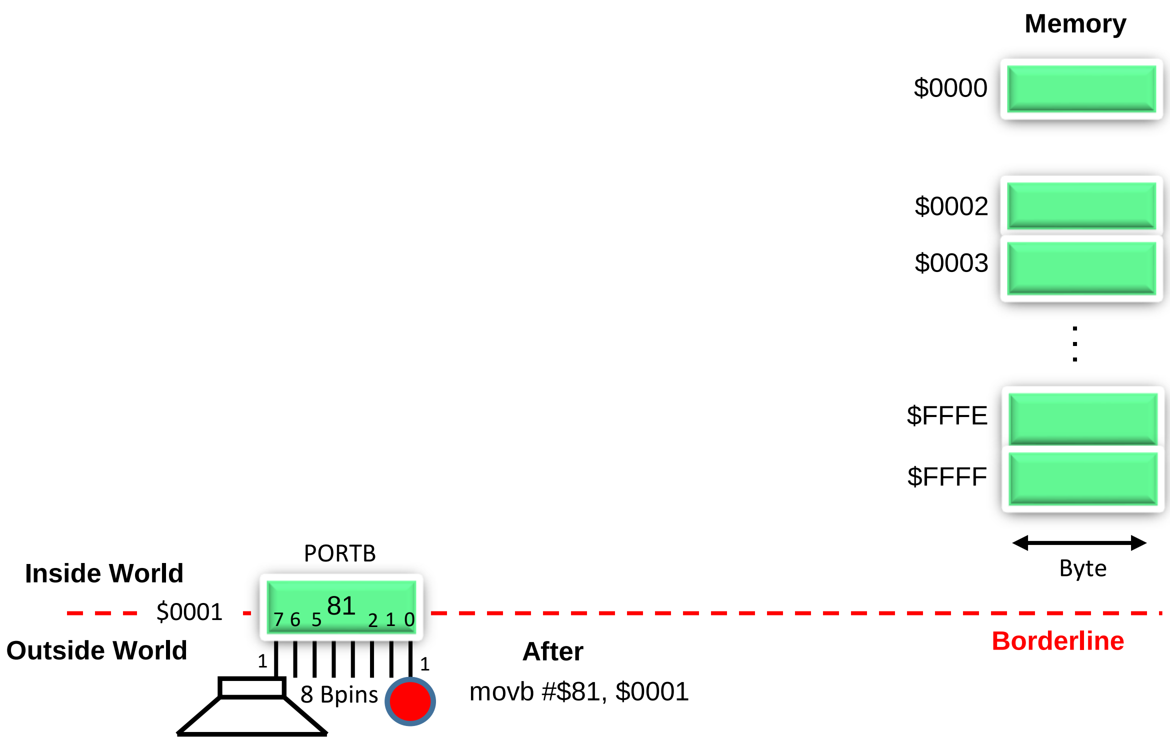


Figure 5. LED turns on when instruction {movb #$81, PORTB} executes

As another example, let us execute instruction {movb #$70, PORTB}, which writes value %0111 0000 in PORTB. Now, Bpin0 is pulled down, turning the LED off while the alarm remains on.

In summary

- The eight outputs of PORTB are tied to 8 pins (call them Bpins), hence exposed to the outside world. (Coming up more ports …)
- Whatever you write in PORTB
  - appears on the Bpins hence becoming visible in the outside world. (Remember from the Digital Design course that the content of a register appears at the output of that register.)
  - will stay there until you overwrite it (similar to any register).

This type of PORT is called *output* PORT, through which you send data to the *outside* world. Soon, you will see *input* PORTs, which work in the opposite direction.

The following section shows how to *read* from an output PORT:

Reading from output PORTB

Consider the following pair of instructions. The first one writes the number $70 in PORTB; the second reads PORTB and places its content ($70) into register B, as shown in Figure 6. The $70 will stay on the eight Bpins as long as the content of PORTB remains the same. If the content of PORTB changes, register B will NOT be affected (unless instruction {ldab PORTB}, for example, executes again).

```
movb	#$70, PORTB
ldab	PORTB
```

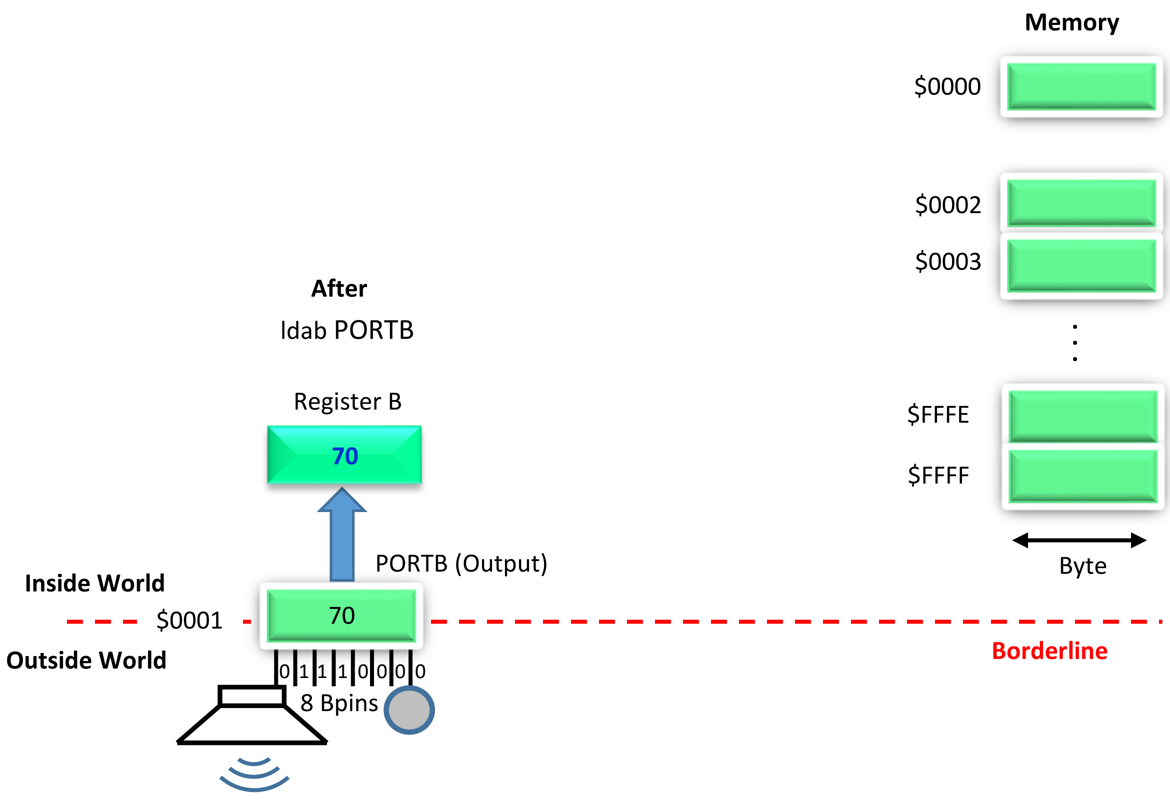


Figure 6. Instruction {ldab PORTB} reads output PORTB and puts the value in register B

Input ports
You can use *output* PORTs to send logic values to the outside world, but not the other way around, so you cannot use output PORTs to read, say, the outside temperature into a CPU's register. To make it happen, you need an *input* PORT through which you can read logic values that the outside world has placed on the PORT's pins. Note that in an input port, the pins are *disconnected* from the outputs of the Data Register; they are under the control of the outside world, as shown in Figure 7 for PORT B, where the outside world has put the value $3E on the Bpins while the content of PORTB is $70 as an example.

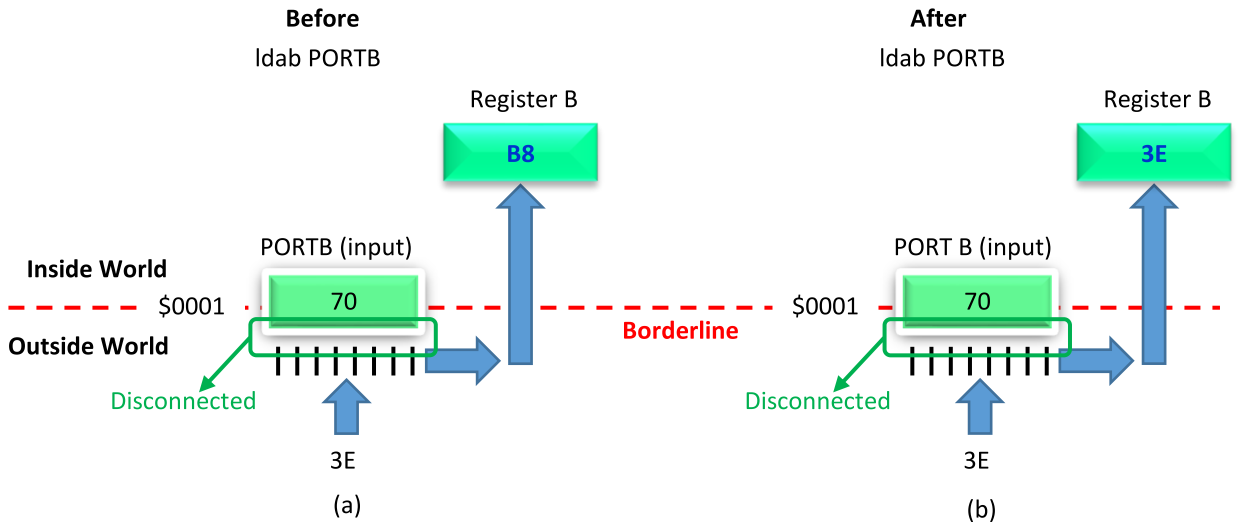


Figure 7. PORT B is input. Bpins are disconnected from the outputs of PORTB. The outside world has placed a $3E on the Bpins.
Register B and PORTB (a) before and (b) after instruction {ldab PORTB} executes.

Question: Is PORT B *input* or *output*?
Answer: "You" decide to configure (or make) each pin of a PORT as an input or output. You make this decision in your program, as you will see shortly.

Example 1. As shown in Figure 7a, let us assume that

- All Bpins are configured as input,
- The content of PORTB (Data Register B) is $70,
- The content of register B is $B8,
- The outside world has placed the environment temperature, $3E, on the Bpins.
- and now instruction {ldab PORTB} executes:

Question: Where does this instruction read from?

Answer: It reads the value placed on the Bpins, not the content of the Data Register of PORT B; in other words, now, literally speaking, PORTB means the pins of PORT B, not the Data Register of PORT B.

Question: Will this instruction change the content of the Data Register of PORT B?

Answer: No, it will not. When this instruction executes, register B will change to $3E; however, the content of PORTB (Data Register of PORT B) will remain $70. See Figure 7b.

Question: *If the outside temperature ($3E) changes, will the content of register B also change?*

Answer: *No, it will not. The content of register B will NOT change unless you read the temperature into register B again (or change register B in any other way) in your program.*

In summary, what you read from an *input* port (using a load instruction, for example)

- is the value that the outside world has placed on the pins of that port, NOT the value sitting in the Port's Data Register.
- is NOT written into the Data Register of that port; it is seated in the destination register specified by the load instruction.

Memory-mapped I/O and Port-mapped I/O

Microprocessors are classified as Memory-mapped I/O and Port-mapped I/O**.** The HCS12 is in the first category, where I/O registers look like regular memory locations; in other words, regular memory locations and I/O registers share the same address space; this is why to access I/O registers, we use the same instructions, such as {movb #$70, $0001}, that we use to access regular memory locations, such as {movb #$70, $3000}. This requires that regular memory addresses and I/O register addresses be mutually exclusive. For example, there is *only* one address, $0001, in the HCS12, and it is the address of PORTB. On the other hand, port-mapped I/O-type microprocessors use different address spaces for the regular memory locations and the I/O registers. Therefore, if the HCS12 were a port-mapped I/O type microcontroller, it would have two locations with, say, address $0001, one for regular memory and one for PORTB. Now the question is how does such a CPU know instruction {movb #$70, $0001} will write in the regular memory at $0001 or in PORTB? And the answer is that this type of microprocessor has two different instructions for these two scenarios, such as {movb #$70, $0001} for regular memory locations and {out #$70, $0001} for PORTB as shown in Figure 8:

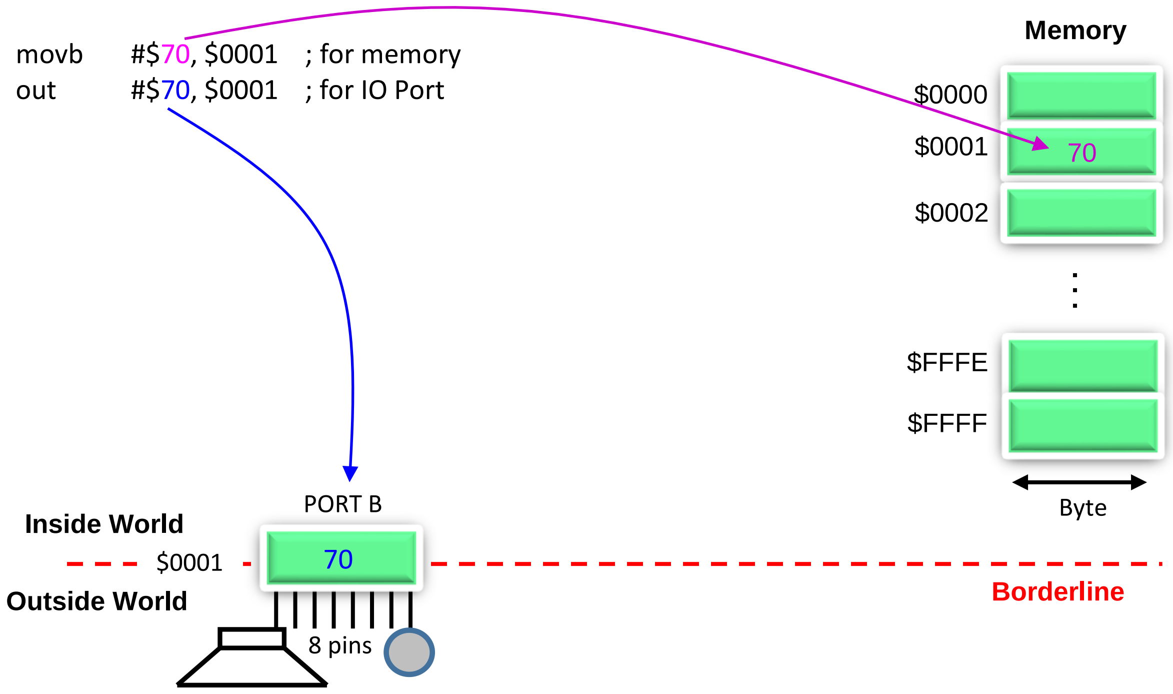


Figure 8. Port-mapped I/O has two address spaces one for regular memory, one for I/O registers

HCS12 ports and port registers:
The HCS12 family's ports are called A, B, J, P, H, K, T, … Different versions of the HCS12 may have different ports. PORT x, where x is the port's name, has eight pins exposed to the outside world, with two exceptions, PORT K is only 7 bits wide and therefore has seven pins, and PORT J is only 4 bits wide and therefore has four pins. We are already familiar with PORT B. Each port has a Data Register located on the borderline between the inside and outside worlds of the microcomputer. Figure 8 shows this for PORT B. The Data Registers of PORTs A, B, and K are called PORTA, PORTB, and PORTK, respectively, and the Data Registers of ports J, P, H, and T are called PTJ, PTP, PTH, and PTT, respectively, as shown in Figure 9.

Each Data Register, x, has its own address and one same-size Data Direction Register, or DDRx for short. DDRx also has its own address. DDRx is not exposed to the outside world. Bit i in DDRx is associated with Pin i of PORT x. A 1 in Bit i of DDRx configures Pin i of PORT x as *output*, and a 0 in Bit i of DDRx configures Pin i of PORT x as *input*.

The table in Figure 9 shows all the information that we usually need for I/O programming on the Dragon 12+ trainer board: In each row, the first column from the left is the port name; the second column shows the port Data Register and its physical address; the third column shows the port Data Direction Register and its physical address, and the last column shows where the port pins are connected to on the Dragon 12+ trainer board:

Port Name	Data Register Name/Address	DDR Name/Address	Dragon 12+ Hardwired
A	PORTA/$0000	DDRA/$0002	Keypad
B	PORTB/$0001	DDRB/$0003	LEDs and 7-Segments
J	PTJ/$0268 (only bits 0,1 6 and 7)	DDRJ/$026A (bits 0,1 6 and 7)	LEDs Enable, …
P	PTP/$0258	DDRP/$025A	7-Segment Enable, …
H	PTH/$0260	DDRH/$0262	Switches, Pushbutton
K	PORTK/$0032 (7 bits wide)	DDRK/$0033	LCD
T	PTT/$0240	DDRT/$0242	Speaker
And More …			

Figure 9. HCS12 I/O ports and where they are connected to on the Dragon 12+ trainer board

In summary,
- PORT x has a Data Register called PORTx or PTx; the right choice depends on x, as shown in Figure 9.
- PORT x has a Data Direction Register called DDRx, as shown in Figure 9.
- PORT x has 4, 7, or 8 pins, as shown in Figure 9.

You will see more registers in the next chapter.
- When a pin is configured as an *output*, that pin is connected to the associated output of the PORT Data Register.

- When a pin is configured as *input*, it is *disconnected* from the PORT Data Register; it becomes under the control of the outside world, i.e., receives its value from the outside world. So, pins are *bidirectional*. You will decide the direction of each pin by programming the associated bit in the port's Data Direction Register (DDR).

Example 2. Instruction {movb #%10010111, DDRA} configures Apins 6, 5, and 3 as input and the rest of the Apins as output. See Figure 10. Remember that when a pin is configured as an output, it is connected to the associated output of the PORT Data Register; otherwise, the pin is disconnected, as shown in Figure 10. This type of instruction would be more readable if the constant was represented in binary.

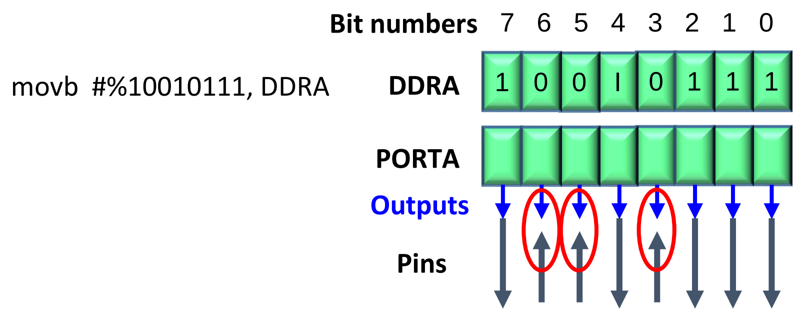


Figure 10. Register DDRA specifies pin directions of PORT A as explained in Example 2

Example 3. Look at the following instructions and comments:

```
movb    #$FF, DDRB      ; configure all Bpins as output

movb    #$55, PORTB     ; write in output PORTB. $55 will appear on the Bpins, becoming visible
                        ; in the outside world.

ldaa    PORTB           ; read from an output port: copy the above $55 from PORTB into register A.
```

The $55 copied into register A is NOT coming from the outside world as PORT B is configured as output.

Example 4. The following instruction configures PORT B as input:

```
movb    #0, DDRB    ; configure all Bpins as input
```

The following instruction reads Bpins; what is read is coming from the "outside world":

```
ldaa    PORTB       ; write in register A whatever the outside world has placed on the Bpins.
                    ; you may also say: read PORTB (but note, you, in fact read the pins of PORT B)
```

*** *Note: This instruction reads the value sitting on the Bpins, not the content of PORTB, the Data Register of PORT B. Figure 12 shows all the possibilities.*

You may also use bset or bclr instructions to configure a subset of the pins of PORTx:

Example 5.

```
bset    DDRA, 1     ; configure Apin0 as output ; the rest of the pins are not affected

bset    PORTA, 1    ; pull Apin0 up; the rest of the pins are not affected

; Note: Apin0 is an abbreviation for Pin0 of PORT A
```

Example 6.

```
movb    #$55, DDRA  ; configure PORT A's even pins as output and odd pins as input
```

Example 7.

```
movb    #$55, PORTB         ; send value $55 to PORTB

movb    #%01010101, PORTB   ; send value $55 to PORTB
```

Example 8. In this example, we will write in an output port, read from the output port, write into an input port, and read from the input port. Review the instructions and the comments rigorously:

```
movb    #$FF, DDRB    ; Configure PORT B as output. The content of PORTB will appear on the Bpins.

movb    #$D5, PORTB   ; Write in PORTB (an output port): PORTB ← $D5. $D5 will appear on the Bpins.

ldab    PORTB         ; Read from PORTB (an output port): register B ← $D5

movb    #0, DDRB      ; Configure PORT B as input.  PORTB's outputs are disconnected from the Bpins.
                      ; $D5 will disappear from the Bpins. Now the pins belong to the outside world.
                      ; The outside world can put logic values on the Bpins.
                      ; Let us say the outside world is now placing a $40 on the Bpins.

ldab    PORTB         ; Read the $40 from the Bpins and put the value ($40) in register B.

movb    #$2D, PORTB   ; Write a $2D in PORTB. $2D will NOT appear on the Bpins as PORT B is still
                      ; configured as input; therefore, the Bpins still belong to the outside world.
```

Now the following instruction changes PORT B direction to out:

**** but before this instruction executes, you must ensure the outside world has removed its value from the Bpins, as illustrated in Figure 11a, which shows only Bpin0 as an example. Figure 11b shows the opposite scenario: You must only change the direction of a pin to out if the pin is open-circuited by the outside world; otherwise, it is, in general, meaningless and may damage the chip.*

```
movb    #$FF, DDRB    ; configure PORT B as output. Value $2D (the content of PORTB) appears on the
                      ; Bpins.
```

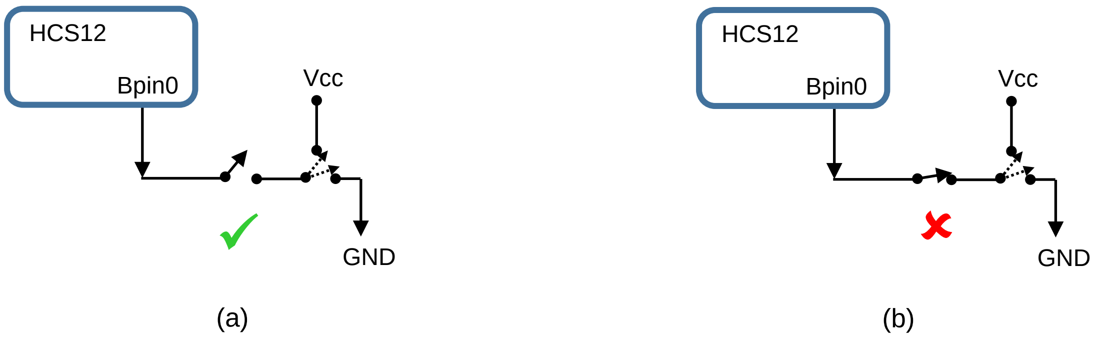


Figure 11. You have to make sure the input pin is open-circuited by the outside world before you change the pin's direction

The different scenarios of writing in or reading from a port are summarized in Figure 12:

In general, some pins of a port may be configured as input and the rest as output. Example 9 is a generalization of Example 8:

	Write in Output Port A	Write in Input Port A	Read from Input Port A	Read from Output Port A
Example	movb #$2D, PORTA	movb #$2D, PORTA	ldab PORTA	ldab PORTA
Comments	$2D is written in Data Register of PORT A; $2D appears on PORT A pins.	$2D is written in Data Register of PORT A; $2D does NOT appear on PORT A pins.	The value placed on **the A pins** (by the outside world) is read and seated in register B.	Content of Data Register of PORT A, in other words, the value sitting on **the A pins**, is read and seated in register B.

Figure 12. Read from and write in ports

Example 9. Go over the following instructions and the comments rigorously:

```
movb    #%0010 0101, DDRB     ; Configure Bpins 5, 2, and 0 as output and Bpins 7, 6, 4, 3, and 1 as
                              ; input. The outside world must not be placing logic values on output
                              ; Bpins 5, 2, or 0 (must leave them open-circuited).

movb    #%1011 0110, PORTB    ; Write binary 1011 0110 in PORTB; this will expose binary 110 on the
                              ; output Bpins 5, 2, and 0 to the outside world.
                              ; The values on input Bpins 7, 6, 4, 3, and 1 are determined by the
                              ; outside world. Let us assume that the outside world has placed binary
                              ; 10010 on Bpins 7, 6, 4, 3, and 1, respectively.

ldaa    PORTB                 ; Copy the (binary) values on the 8 pins into Register A: A ← 10101100
                              ; Note that 10010 come from the outside world, and 110 come from
                              ; PORTB, so we can say all the 8 bits come from the Bpins.
                              ; Now let us say the outside world open-circuits Bpins 7, 6, 4, 3, and 1
                              ; as well.

movb    #$FF, DDRB            ; Configure all the Bpins as output. So, binary 1011 0110 will be sitting
                              ; on the Bpins.

ldaa    PORTB                 ; Copy Bpins into Register A: A ← 1011 0110, or, copy PORTB into
                              ; Register A, but here by PORTB, we mean pins of PORT B.
```

Conclusion: *Instruction {ldaa PORTB} always reads the Bpins, whether PORT B is configured as input, output, or some pins as input and some as output.*

How to create delays:

One-Hz square-wave generator: How do you create a one-Hz square wave on, say, Apin0? You should pull Apin0 up, wait for 0.5 seconds, then pull it down and wait for 0.5 seconds again, and repeat this cycle forever. Now the question is, how can you generate a 0.5-second delay?

Each instruction takes a certain number of *E-clock cycles*, or *E-cycles* for short. The E-clock is one of the clock signals of the HCS12 microcontroller. The frequency of this clock (on the Dragon 12+ board) is 24 MHz, or its period is approximately 41.7 nanoseconds. Therefore, a 3-cycle instruction, for example, needs 41.7 x 3 = 125 nanoseconds. You may look up the number of E-cycles in the *Access Detail* column in the CPU reference manual. You may also use the CodeWarrior software to figure out how many E-cycles an instruction needs: go to the single-stepping mode, check the upper left corner of the Register pane of the True-Time Simulator and Real-Time Debugger window before and after the instruction executes; the difference between the two readings equals the number of E-cycles for that instruction. In general, the delay caused by executing a group of instructions would be:

Delay = (Total number of E-cycles that those instructions need to execute) x E-clock period

Example 10. Calculate the total delay created by the following code: (dbne needs 3 E-cycles)

```
      ldx     #24000   ; initialize loop counter
wait: dbne    X, wait  ; 24000 x 3 E-cycles
```

Total delay ≈ 24000 (No of iterations) x 3 (E cycles per each dbne) x 41.7 ns (E cycle time) ≈ 3 ms delay

Example 11. Create a 300-ms delay:

If the number of iterations in the above loop is multiplied by 100, we will get a delay of approximately 3 x 100 = 300 ms.

Number of iterations = 24000 x 100 = 2 400 000

2 400 000 does not fit in one 16-bit register (2 400 000 > 64K), so we use the following nested loops:

```
        ldab #100
outer:  ldx   #24000      ; 100 iterations in the outer loop
inner:  dbne X, inner     ; 24000 x 3 E-cycles in the inner loop
        dbne B, outer
```

Example 12. Obtain the delay created by the following code. The number of E-cycles for each instruction is shown next to that instruction:

```
        ldx   # 20000
wait:   psha              ; (2 E-cycles),
        pula              ; (3 E-cycles)
        nop               ; (1 E-cycle)
        nop               ; (1 E-cycle)
        nop               ; (1 E-cycle)
        nop               ; (1 E-cycle)
        dbne X, wait      ; (3 E-cycles)
```

Total number of E-cycles = 2 + 3 + 1 + 1 + 1 + 1 + 3 = 12

Total delay = (12 E-cycles) x 20000 x 1/24M = 10 000 us (microseconds) = 10 ms

Example 13. Using the above code, write subroutine delay10Y, which takes Y as an input and generates a 10 x Y ms delay, where 0 < Y. For example, if Y = 50, the subroutine will cause a delay of 10 x 50 = 500 ms.

```
delay10Y:
        pshx
        pshy
        pshc
wait10Y: ldx   # 20000
wait10:  psha             ; (2 E-cycles),
        pula                ; (3 E-cycles)
        nop                 ; (1 E-cycle)
        nop                 ; (1 E-cycle)
        nop                 ; (1 E-cycle)
        nop                 ; (1 E-cycle)
        dbne X, wait10      ; (3 E-cycles)
        dbne Y, wait10Y
        pulc
        puly
        pulx
        rts
        end
```

Example 14. Write a subroutine to generate a 50-microsecond delay:

Total number of E-cycles = 50 us/41.7 ns = 1200

So, we can use 400 x dbne instructions in a loop:

```
Delay50u:   pshx
            pshc
            ldx     #400      ; initialize loop counter (an 8-bit register is not enough)
    wait:   dbne    X, wait   ; wait here for 400 x 3 E-cycles
            pulc
            pulx
            rts
```

Example 15. Write a program to generate a 10-KHz square wave with a 50% duty cycle on Apin0:

We need to pull up Apin0, wait for 50 us, pull down Apin0, and wait for 50 us again. And repeat this cycle forever. The waveform is shown in Figure 13:

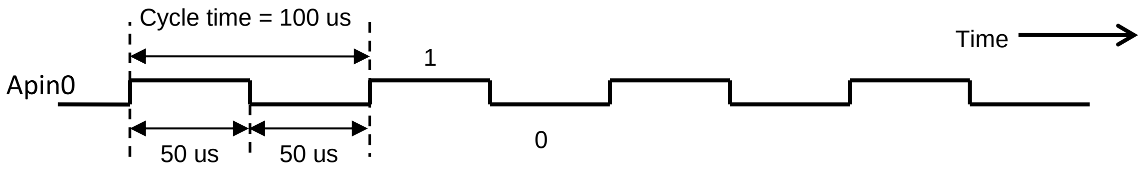


Figure 13. 10-KHz square wave in Example 15

Here is the code to generate the above waveform:

```
            bset  DDRA, #1      ; configure Apin0 (Pin 0 of PORT A) as output
forever:    bset  PORTA, #1     ; pull Apin0 up
            jsr   delay50u      ; wait for 50 us (this subroutine was written in Example 14.)
            bclr  PORTA, #1     ; pull Apin0 down
            jsr   delay50u      ; wait for 50 us
            bra   forever
```

Eight individual LEDs on the Dragon 12+ trainer board

A diode is a two-terminal circuit element. The terminals are called the *anode* and the *cathode*. The circuit symbol for a diode is shown in Figure 14a. When Voltage of Anode – Voltage of Cathode) > ≈ 0, the diode turns on, which means that, ideally speaking, it is equivalent to a one-way wire, as shown in Figure 14b. Now the current flows from the anode to the cathode. If the voltage difference {Voltage of Anode – Voltage of Cathode} is not high enough, the diode turns off, which means that, ideally speaking, the diode is equivalent to a disconnected switch, as shown in Figure 14c. Now the current through the diode ≈ 0. A light-emitting diode, or LED for short, is a special diode: when an LED turns on, it emits light; otherwise, it does not. A circuit symbol for an LED is shown in Figure 14d.

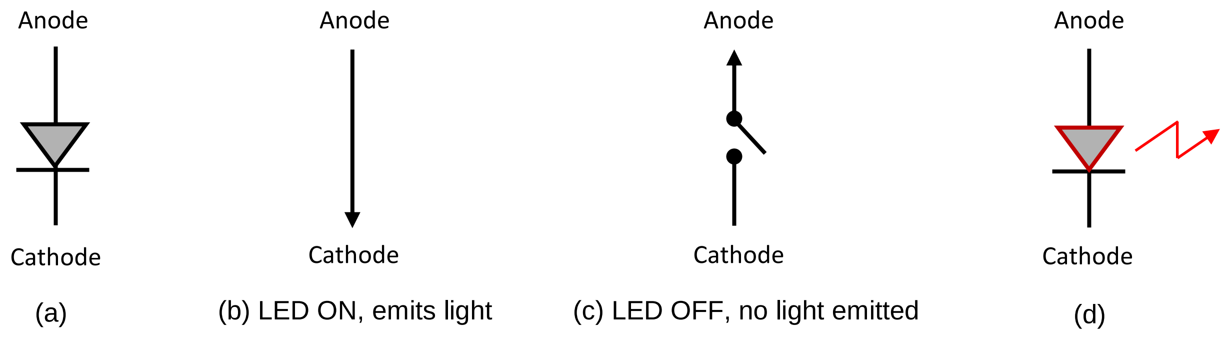
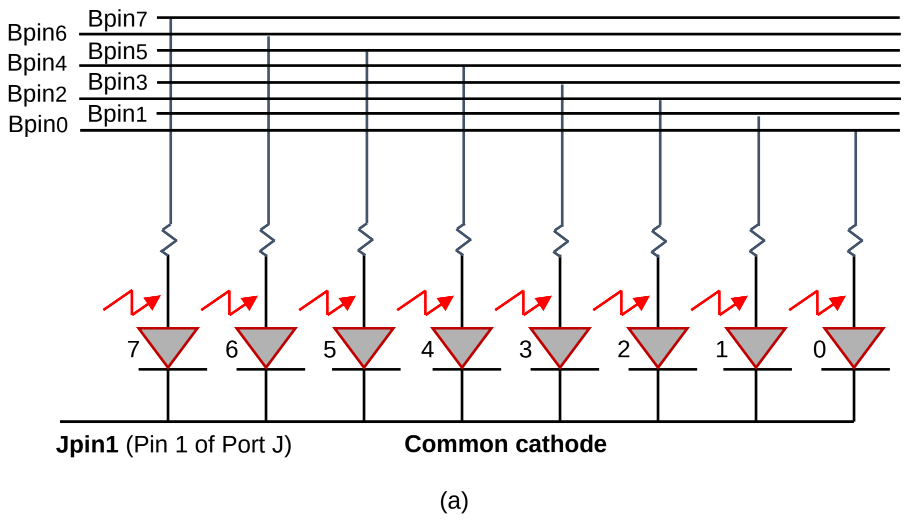


Figure 14. (a) diode's circuit symbol, (b) the ideal model for diode ON, (c) the ideal model for diode OFF, (d) the LED's circuit symbol.

There are eight individual LEDs on the Dragon 12+ board, as shown in Figure 15a. The eight Bpins drive the eight anodes of the LEDs. The cathodes of all the LEDs are tied together and driven by Jpin1, Pin 1 of PORT J[2]. Therefore, if Jpin1 (the common cathode) is pulled up, all the LEDs will turn off unconditionally. This scenario is shown in the last two rows of the truth table of Figure 15b. To turn on an LED, the common cathode must be pulled down, and the anode of the LED must be pulled up, as shown in the second row of the truth table; otherwise, the LED will turn off, which is shown in the first row of the truth table.



$Jpin_1$	$Bpin_n$ $0 \le n \le 7$	LED_n $0 \le n \le 7$
0	0	OFF
0	1	ON
1	0	OFF
1	1	OFF

(b)

Figure 15. (a) The eight individual LEDs on the Dragon 12+ trainer board, (b) truth table.

[2] Some Dragon 12+ boards may also have other settings.

Example 16. Write a program to blink LED2 @ 1 Hz:

```
                ; Initialization
                bset    DDRJ, 2         ; configure Jpin1 as output
                bclr    PTJ, 2          ; enable LEDs
                bset    DDRB, 4         ; configure Bpin2 as output
                ldy     #50             ; to get a 500-ms delay
                ; infinite loop
forever:        bset    PORTB, 4        ; turn LED2 ON
                jsr     delay10Y        ; wait for 10 x 50 = 500 ms
                bclr    PORTB, 4        ; turn LED2 OFF
                jsr     delay10Y        ; wait for 10 x 50 = 500 ms
                bra     forever         ; do it forever
                end
```

We may also use the *read, modify, write algorithm*:

```
                ; Initialization
                bset    DDRJ, 2         ; configure Jpin1 as output
                bclr    PTJ, 2          ; enable LEDs
                bset    DDRB, 4         ; configure Bpin2 as output
                ldy     #50             ; to get a 500-ms delay
                ; infinite loop
forever:        ldaa    PORTB           ; read PORTB
                eora    #4              ; modify: toggle Bit2 of PORTB
                staa    PORTB           ; write: toggle LED2
                jsr     delay10Y        ; wait for 10 x 50 = 500 ms
                bra     forever         ; do it forever
                end
```

Subroutine delay10Y, written in Example 13, receives a value in register Y and generates a 10Y-ms delay.

Example 17. Moving LED: Write a program to turn one LED on at a time from left to right and then from right to left forever:

As shown in Figure 15, on the Dragon 12+ board, each LED turns on with logic 1, provided that the common cathode is pulled down. To turn one LED at a time on in one direction and then in the opposite direction, we need to send the following patterns to PORTB in the same order:

1000 0000   0100 0000   0010 0000   0001 0000   0000 1000   0000 0100   0000 0010   0000 0001   0000 0010   0000 0100   0000 1000   0001 0000 0010 0000   0100 0000 start over …

```
; Data Segment (hexadecimal equivalents to the above binary patterns)
table:      dc.b $80, $40, $20, $10, $08, $04, $02, $01, $02, $04, $08, $10, $20, $40

            ; Initialization
            movb    #$FF, DDRB      ; configure PORT B as output
            bset    DDRJ, $02       ; configure Jpin1 as output
            bclr    PTJ, $02        ; enable LEDs to light up
```

```
          ldy     #50            ; to get a 500-ms delay
          ; infinite loop
forever:  ldaa    #14            ; initialize loop counter, A
          ldx     #table         ; X becomes pattern pointer
loop:     movb    1, X+, PORTB   ; light up next LED
          jsr     delay10Y       ; wait for 10 x 50 = 500 ms
          dbne    A, loop        : update loop counter; if not 0, continue the current sequence
          bra     forever        ; otherwise, start a new sequence.
```

PORT H, eight DIP switches, and four pushbuttons on the Dragon 12+ trainer board
Eight **DIP switches** drive PORT H on the Dragon 12+. Additionally, there are four PBs, **pushbuttons**, in parallel with the four least significant DIP switches, as shown in Figure 16:

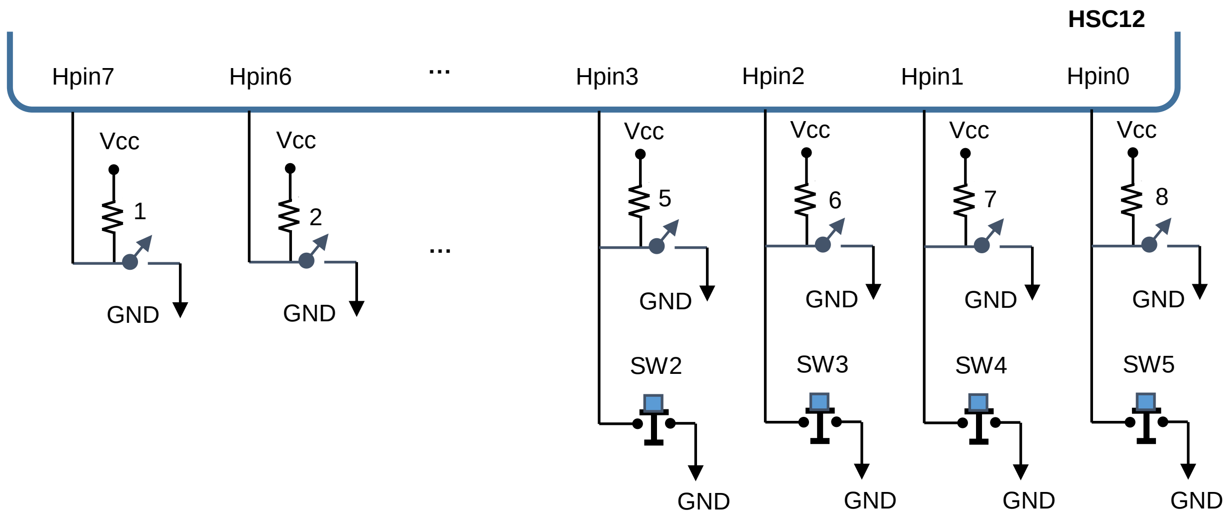


Figure 16. PORTH on Dragon 12+ board driven by eight DIP switches and four pushbuttons.

Both *pushbuttons* and *PORTB pins* may be abbreviated as PB. What is meant should be clear from the context; otherwise, Bpin, the other abbreviation, is used.

To make indexing more readable, for each DIP switch and pushbutton, we use the index of the pin that the DIP switch or the pushbutton is connected to, as shown in Figure 17; e.g., PB0 and DIP0 are connected to Hpin0 (Pin0 of PORT H):

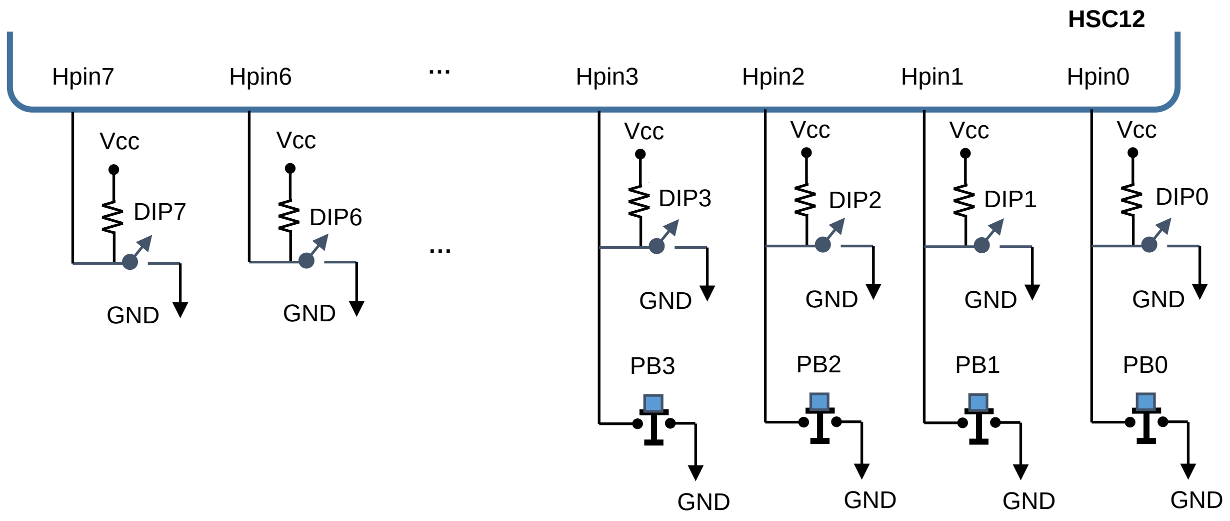


Figure 17. Pin numbers are used to index the DIP switches and the pushbuttons.

When a DIP switch is slid up, it will be disconnected. The pushbuttons are normally disconnected. The truth table for the four right-most pins is shown in Figure 18:

DIP Switch i $0 \le i \le 3$	**Pushbutton i** $0 \le i \le 3$	**Hpin i** $0 \le i \le 3$
Slid down (0)	X	0
X	Pressed (0)	0
Slid up (1)	Released (1)	1

Figure 18. The truth table for the four right-most Hpins (pins of PORT H)

No pushbuttons are connected to the four left-most pins of PORT H, as shown in Figure 16. So, the voltage level on any of these pins is only determined by the corresponding DIP switch: if a switch is slid up, the pin (connected to that switch) will be pulled up, otherwise, pulled down.

Example 18. Write a program to turn LED2 (driven by Bpin2) ON when PB2 is pushed. LED2 should remain ON until PB2 is released and pushed again. The hardware is shown in Figure 19:

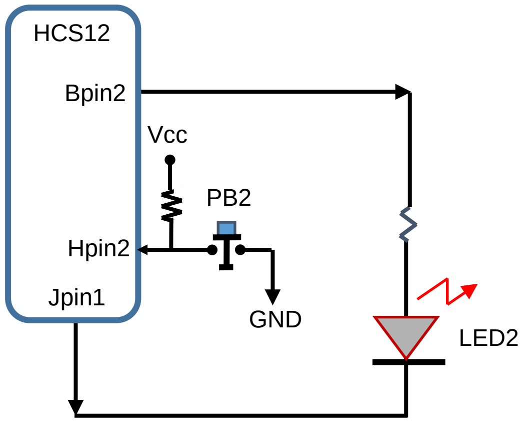


Figure 19. Hardware used in Example 18.

```
; Initialization
bset   DDRJ, $2          ; configure Jpin1 as output
bclr   PTJ, $2           ; enable LEDs to light up
movb  #$FF, DDRB         ; configure PORT B as output
movb  #0, PORTB          ; turn LEDs OFF
movb  #$F, DDRP          ; configure Ppin3-0 as output
movb  #$0F, PTP          ; turn 7-segment displays OFF
movb  #0, DDRH           ; configure PORT H as input
```

```
; infinite loop
forever:
checkPsh:
     brset     PTH, 4, checkPsh     ; wait until pushbutton-2 is pressed (Check if the pushbutton is pressed)

     ldab      PORTB                ; read PORT B

     eorb      #4                   ; XOR to invert bit 2

     stab      PORTB                ; toggle LED2

checkRel:
     brclr     PTH, 4, checkRel     ; wait until pushbutton-2 released (Check if pushbutton-2 is released)

     bra       forever
```

Do you know that there is; however, a major issue called *switch bounces* with this code? If not, then go over the rest of this section rigorously:

A pushbutton-based manual pulse generator and the *ideal* behavior of the pushbutton are shown in Figure 20:

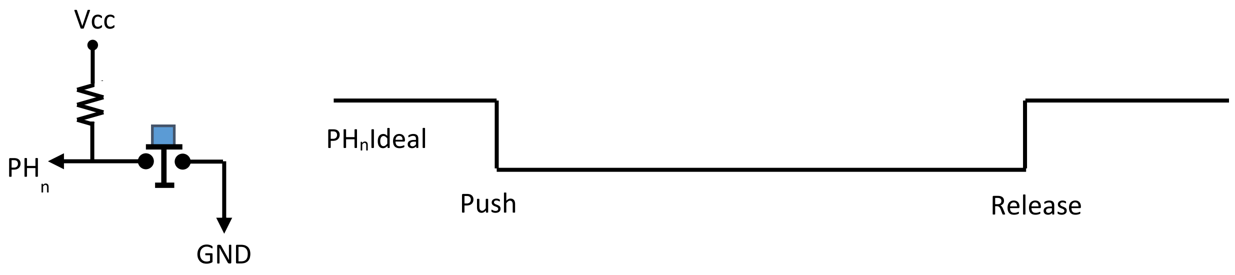


Figure 20. Ideal behavior of a pushbutton

The real behavior of a pushbutton (and any mechanical switch) is shown in Figure 21. When the pushbutton is pressed, and its wiper hits the two terminals, the wiper bounces (i.e., makes and breaks contact) a few times before settling (*contact-bounce* problem). And this creates *unwanted* and possibly *problematic* pulses, as shown in Figure 21. Unwanted pulses may also be generated when the pushbutton is released. See Figure 21. These pulses (bounces) are too narrow for us to see directly on a, say, LED, while they are wide enough for microcontrollers to sample and detect. So, we need to clean up PH_n^{real}, or the microcontroller will see one keypress as multiple presses!

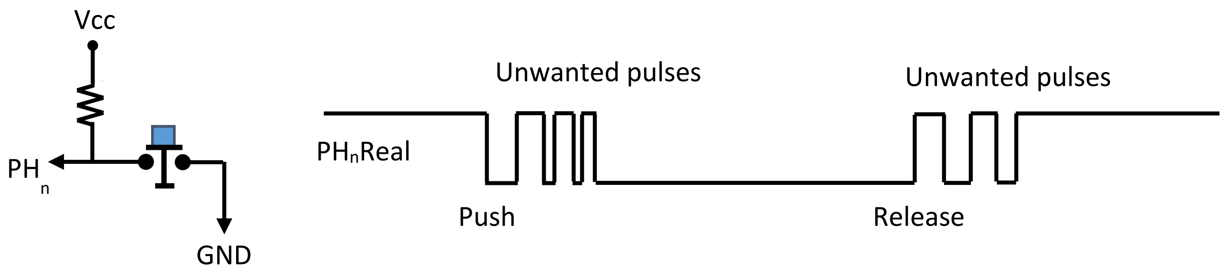


Figure 21. Real behavior of a pushbutton

There are hardware and software techniques to resolve the contact bounce issue. In Figure 22, an RS latch debounces a single-pole, double-through (SPDT) pushbutton. This circuit is also able to hide the possible bounces that could occur when the button is released:

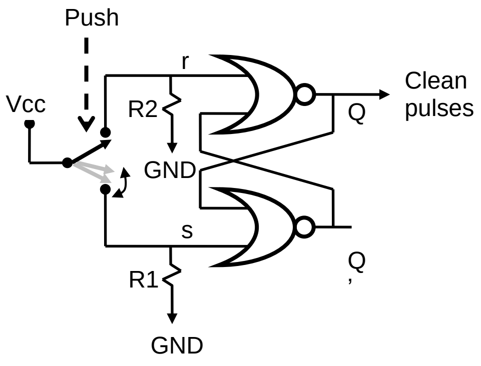


Figure 22. RS-latch-based debouncer

Here are two well-known software algorithms for debouncing:

- Wait and See:

Once the first transition in the output signal (from a, say, pushbutton) is detected, wait and then check it again; if the signal still has the same value, assume that the bounces have gone away; otherwise, ignore the signal transition.

- Wait and Go:

Once the first transition in the output signal (from a, say, pushbutton) is detected, wait to let the bounces go away, and then continue. See Figure 23:

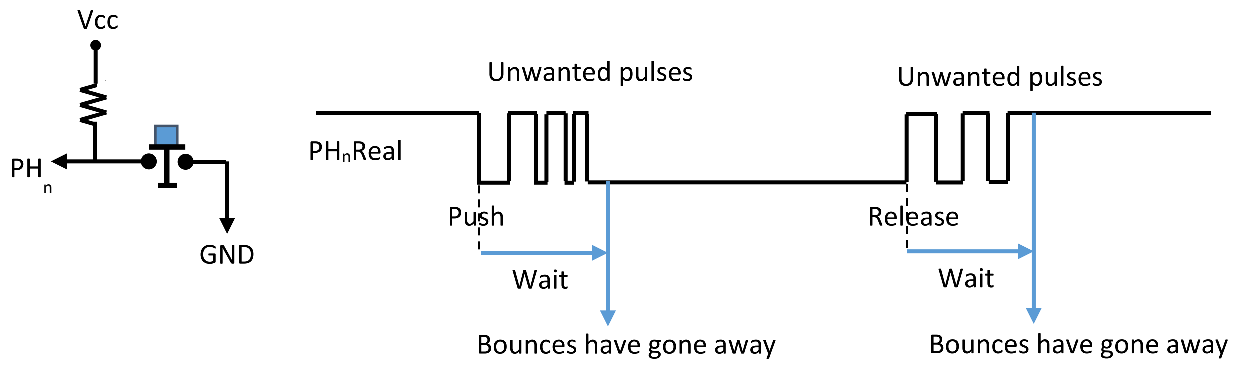


Figure 23. Graphical representation of the wait-and-go algorithm

Example 18 (Cont'd)
Back to Example 18, we are going to debounce the pushbutton using the wait-and-see algorithm:

```
; Initialization (same as before)
bset    DDRJ, $2     ; configure Jpin1 as output
bclr    PTJ, $2      ; enable LEDs to light up
movb    #$FF, DDRB   ; configure PORT B as output
movb    #0, PORTB    ; turn LEDs OFF
movb    #$F, DDRP    ; configure Ppin3-0 as output
```

```
     movb      #$0F, PTP      ; turn 7-segment displays OFF
     movb      #0, DDRH       ; configure PORT H as input

; infinite loop
forever:
checkPsh:
     brset     PTH, 4, checkPsh   ; wait until PB2 is pressed (check if PB2 is pushed)
     jsr       delay              ; wait to let the bounces go away
     brset     PTH, 4, checkPsh   ; if not pressed, assume that the PB2 is not pressed at all (see)

     ; otherwise, PB2 is pushed, toggle LED2
     ; so, read PORTB, invert bit 2, and write back
     ldab      PORTB              ; read status of LEDs
     eorb      #4                 ; XOR to invert bit 2
     stab      PORTB              ; toggle LED2
checkRel:
     brclr     PTH, 4, checkRel   ; wait until PB2 is released (check if the PB2 is released)
     jsr       delay              ; wait to let the bounces go away
     brclr     PTH, 4, checkRel   ; if not released, assume that the PB2 is not released at all (see)
     bra       forever
```

***

The algorithm will change to the “wait and go” should the following two instructions be commented out:

```
     brset     PTH, 4, checkPsh   ; if not pressed, assume that the PB2 is not pressed at all (see)

     brclr     PTH, 4, checkRel   ; if not released, assume that the PB2 is not released at all (see)
```

Four 7-segment displays on the Dragon 12+ trainer board

A 7-segment display comprises seven LEDs as seven segments (to display a hexadecimal digit) plus one more LED for the hexadecimal point or usually called the decimal point. Seven-segment displays are available in two different forms: common *anode* and *common cathode*. The circuit symbol of common anode displays is shown in Figure 24a, where the top segment is called a, and the other segments are named in alphabetical order if you move clockwise. The middle segment is g, and the decimal point is h. More details of this display are illustrated in Figure 24b. The eight LEDs are shown in alphabetical order from the right to left. As you see, the anodes of all the LEDs are connected. This node is called the *common anode*. To turn on an LED, the common anode has to be pulled up; then if the cathode of an LED is pulled down, that LED will light up.

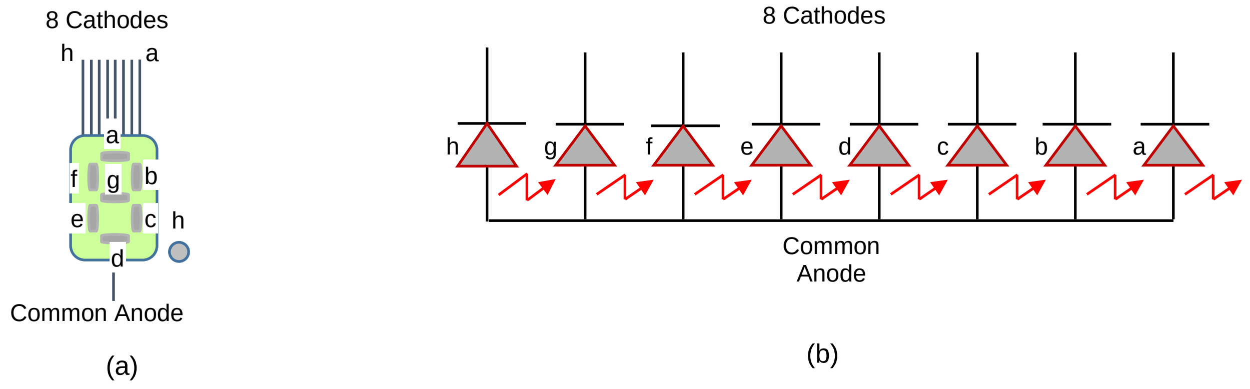


Figure 24. Common-anode 7-segment display (a) circuit symbol, (b) schematic.

Figure 25 shows the truth table of hexadecimal to common-anode 7-segment decoders. Note that each LED turns on with a logic 0 (provided that the common anode is pulled up) and turns off with a logic 1. Digit 5 (with no decimal point) is displayed in this figure as an example.

Hexadecimal Digit	h	g	f	e	d	c	b	a	Hexadecimal 7-segment code words
0	1	1	0	0	0	0	0	0	C0
1	1	1	1	1	1	0	0	1	F9
2	1	0	1	0	0	1	0	0	A4
3	1	0	1	1	0	0	0	0	B0
4	1	0	0	1	1	0	0	1	99
5	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**92**
6	1	0	0	0	0	0	1	0	82
7	1	1	1	1	1	0	0	0	F8
8	1	0	0	0	0	0	0	0	80
9	1	0	0	1	0	0	0	0	90
a	1	0	0	0	1	0	0	0	88
b	1	0	0	0	0	0	1	1	83
c	1	0	1	0	0	1	1	1	A7
d	1	0	1	0	0	0	0	1	A1
E	1	0	0	0	0	1	1	0	86
F	1	0	0	0	1	1	1	0	8E

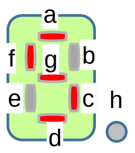
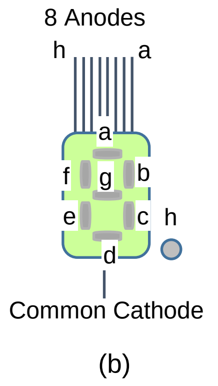


Figure 25. Truth table of hexadecimal to common-anode 7-segment decoders: each LED turns on with a logic 0. The decimal point is turned off in all rows. Digit 5 is displayed as an example.

The circuit symbol of common cathode displays is shown in Figure 26a, and more details about it are illustrated in Figure 26b. The LEDs are numbered similarly to those in common-anode displays. As you see, the cathodes of all the LEDs are connected. (This is similar to the eight individual LEDs available on this trainer board.) To turn on an LED, the common cathode has to be pulled down; then, if the anode of an LED is pulled up, that LED will light up.



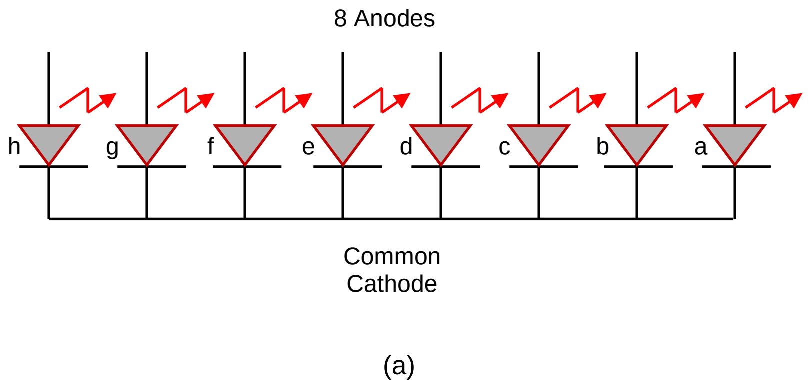


Figure 26. Common-cathode 7-segment display (a) circuit symbol, (b) schematic

Figure 27 shows the truth table of hexadecimal to common-cathode 7-segment decoders. Note that each LED turns on with a logic 1 (provided that the common cathode is pulled down) and turns off with a logic 0. Digit 9 is displayed in this figure as an example.

Hexadecimal digit	h	g	f	e	d	c	b	a	Hexadecimal 7-segment codewords
0	0	0	1	1	1	1	1	1	3F
1	0	0	0	0	0	1	1	0	06
2	0	1	0	1	1	0	1	1	5B
3	0	1	0	0	1	1	1	1	4F
4	0	1	1	0	0	1	1	0	66
5	0	1	1	0	1	1	0	1	6D
6	0	1	1	1	1	1	0	1	7D
7	0	0	0	0	0	1	1	1	07
8	0	1	1	1	1	1	1	1	7F
9	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**6F**
a	0	1	1	1	0	1	1	1	77
b	0	1	1	1	1	1	0	0	7C
c	1	1	0	1	1	0	0	0	58
d	0	1	0	1	1	1	1	0	5E
E	0	1	1	1	1	0	0	1	79
F	0	1	1	1	0	0	0	1	71

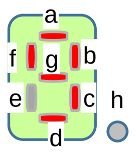


Figure 27. The truth table of hexadecimal to common-cathode 7-segment decoders: each LED turns on with a logic 1. The decimal point is turned off in all rows. Digit 9 is displayed as an example.

There are four common-cathode 7-segment displays on the Dragon 12+ trainer board, as shown in Figure 28. As you see, PORTB drives the eight anodes of each display: Bpin0 drives segment a of the four displays, Bpin1 drives segment b of the four displays, all the way up to Bpin7, which drives the decimal point of the four displays. Also, note that the common cathodes of the four displays from left to right are driven by Ppin0 through Ppin3, respectively.

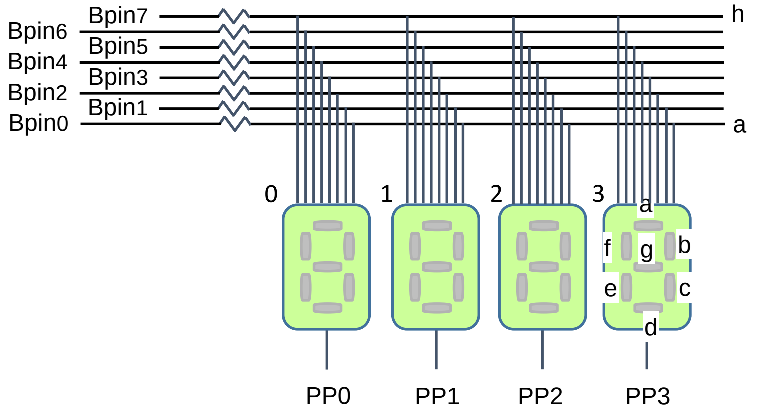


Figure 28. Four x 7-segment displays on the Dragon 12+ trainer board.

Example 19. Consider the cathode and anode signals applied to the four x 7-segment display, as shown in Figure 29. The four common cathodes are tied to logic 1; therefore, all the displays will turn off.

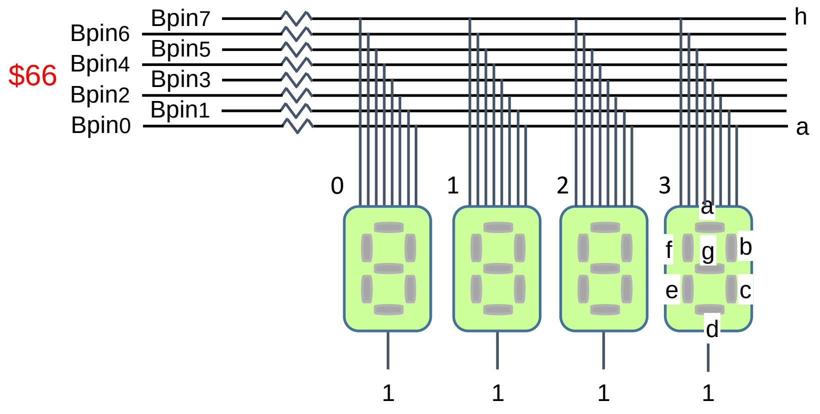


Figure 29. Four x 7-segment displays in Example 19: all four displays are turned off.

Example 20. Consider the cathode and anode signals applied to the four x 7-segment display, as shown in Figure 30. A hexadecimal $66 is applied to the anodes of all the displays. As shown in Figure 27, $66 is the 7-segment codeword of digit 4. On the other hand, only the common cathode of display #1 is pulled down. Therefore, digit 4 will be displayed on this display as shown in Figure 30:

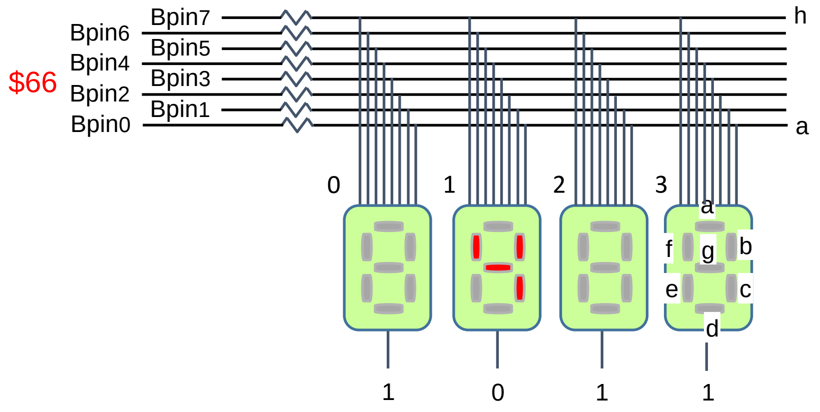


Figure 30. Four x 7-segment display in Example 20.

Example 21. More examples, similar to the previous one, are shown in Figure 31. Go over them carefully:

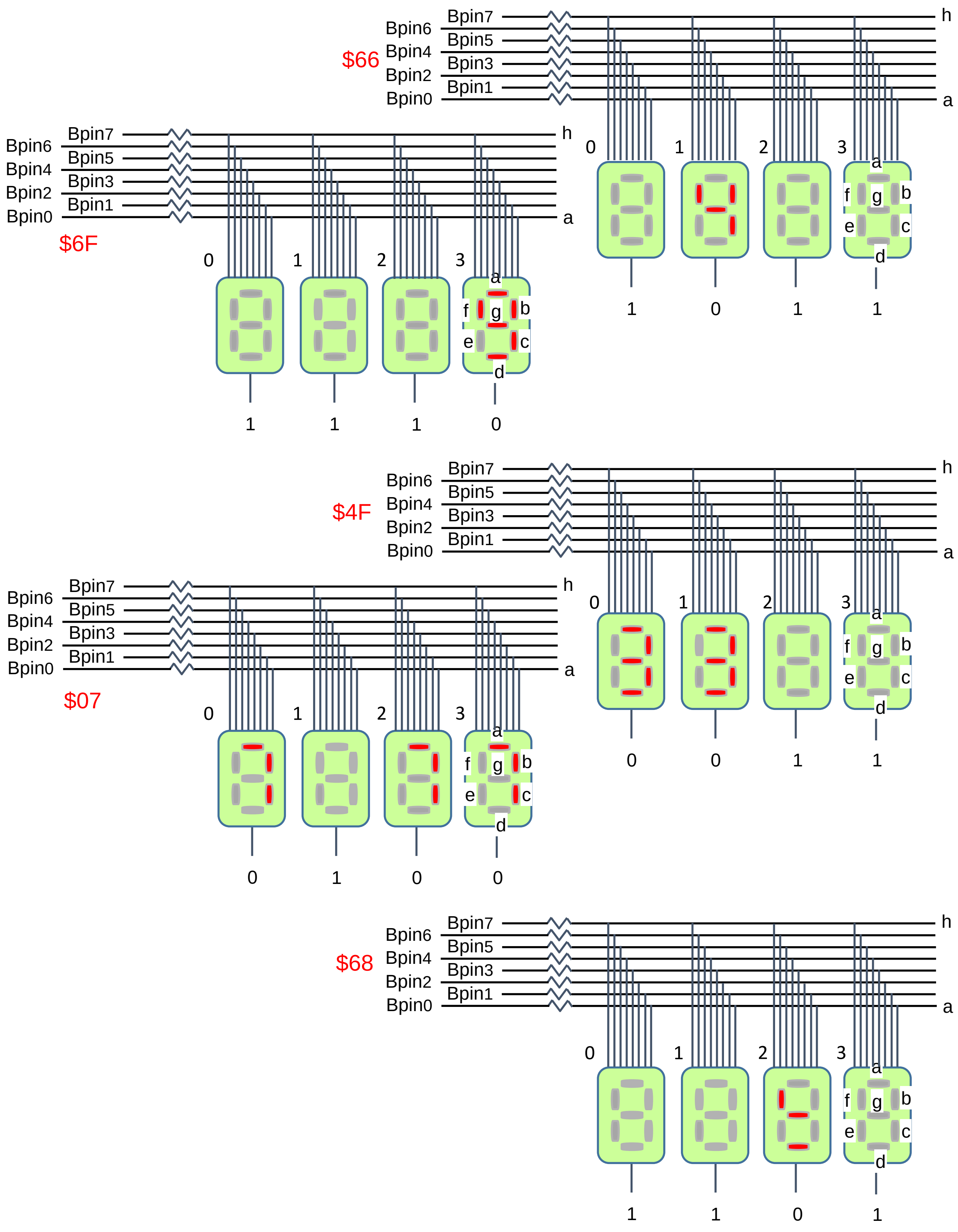


Figure 31. Different digits displayed on the four 7-segment displays in Example 21.

Example 22. Write a program to display digit 5 on display #0, as shown in Figure 32:

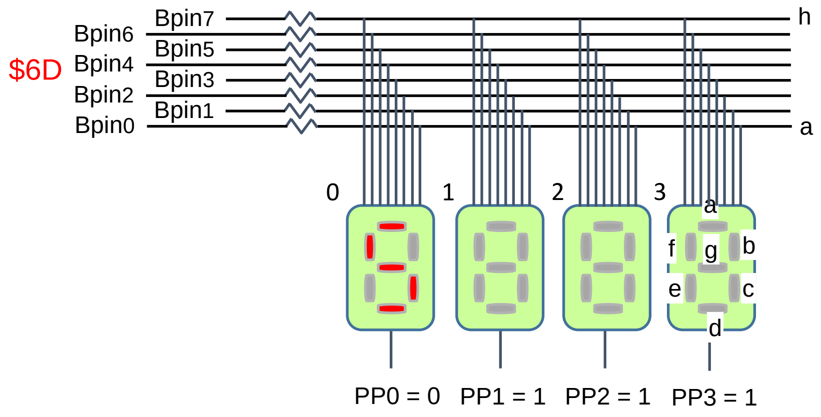


Figure 32. Four x 7-segment display in Example 22.

```
movb    #$FF, DDRB              ; configure PORT B as output
movb    #%F, DDRP               ; configure Ppin0:3 as output
movb    #$6D, PORTB             ; PORTB ← digit 5, see Figure 27
movb    #%1110, PTP             ; enable Display 0
```

In this example PORTB and PTP are not overwritten and stay $6D and $F, respectively, forever; therefore, we do not need an infinite loop.

Multiplexing/Demultiplexing data on four 7-segment displays using pure hardware
As you see in Figure 32 (and similar figures), PORTB drives all the 7-segment displays on the Dragon 12+ board; therefore, the *same* 8-bit value on PORTB pins is sent to all the 7-segment displays. Does this mean we cannot display four digits together on these four displays? Considering the following fact, the answer is no, it does not:

The photoreceptors in our retinae are unable to perceive the flashing LEDs at frequencies higher than about 50 Hz.

Example 23. A digital circuit (pure hardware) that displays the pattern 3456 on four common cathode 7-segment displays is shown in Figure 33. The 7-segment codewords $4F, $66, $6D, and $7D of the four digits 3, 4, 5, and 6, respectively, are applied to inputs 0, 1, 2, and 3, respectively, of a 4-input 8-bit multiplexer. The output of the multiplexer drives the eight anodes of the four displays. These eight interconnects or wires are called a *bus* (as covered in Digital Design courses.) A bus is one or more interconnects (wires) shared by two or more data producers (transmitters) or two or more data consumers (receivers). To let two or more transmitters share one or more output wires (interconnects), we need to perform *multiplexing*; a multiplexer does multiplexing. We need to perform demultiplexing to let two or more receivers share one or more input wires (interconnects). A binary decoder can demultiplex multiplexed data, as you will see shortly. The four displays are the receivers in this example. The transmitters are the 4 x 8 slide switches that drive the 4 x 8-bit inputs of the multiplexer, as shown in Figure 33. A 2-bit free-running binary counter drives the select inputs of the multiplexer and the select inputs of a 2-to-4 active-low binary decoder.

The counting sequence of the 2-bit binary counter is … 11, 00, 01, 10, 11, 00, 01, … Therefore, the output values of the multiplexer will be … $7D, $4F, $66, $6D, $7D, $4F, $66, … This is the concept of *multiplexing*: the output of the multiplexer (bus) is shared by some transmitters (data producers), one at a time. In this example, the sharing is *periodic*, and the number of transmitters is 4, each consisting of eight slide switches.

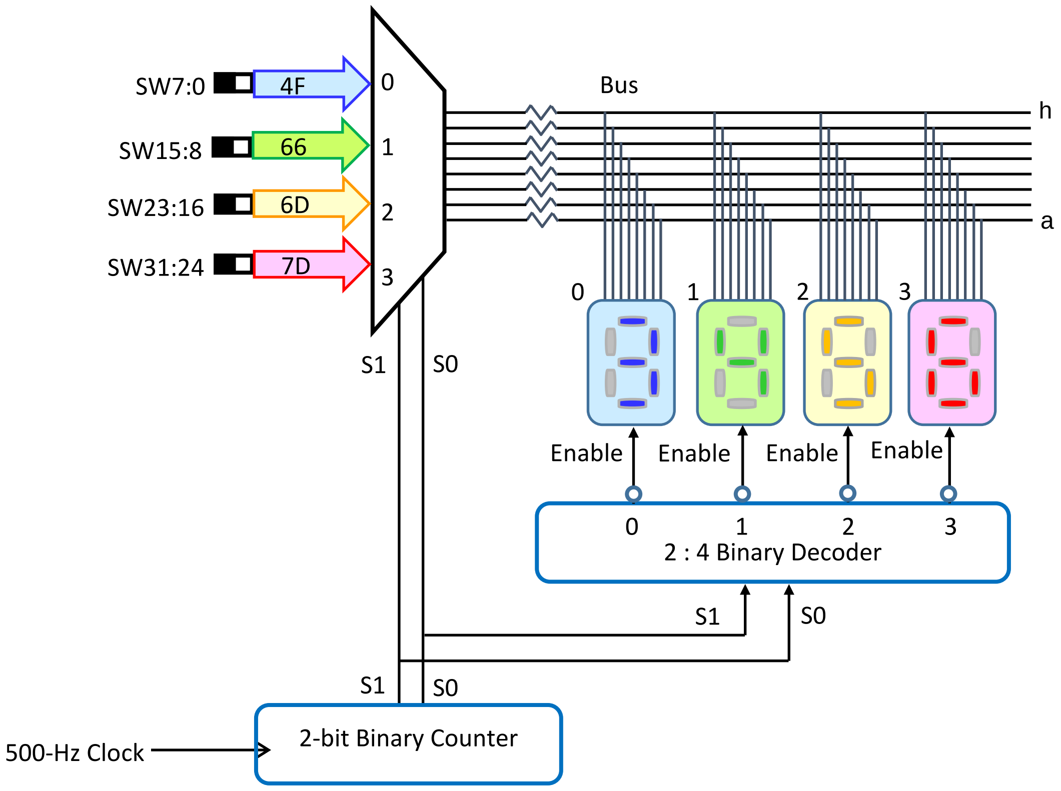


Figure 33. 4-digit multiplexing/demultiplexing in Example 23

The output of the binary counter is also applied to a 2-to-4 active-low binary decoder, and the four outputs of the decoder drive the common cathodes of the four 7-segment displays. This way, it is guaranteed that the value on the bus is delivered to only the correct display (data consumer) at any time; this is the concept of *demultiplexing*: the bus is shared by some (here 4) receivers (data consumers), one at a time, to deliver the right data to the right receiver. Let us take a closer look at what is happening here:

Since the select inputs of the multiplexer and the decoder are tied together,

- when input 0 of the multiplexer is chosen, output 0 of the decoder is asserted; so, Display 0 will turn on;
- when input 1 of the multiplexer is chosen, output 1 of the decoder is asserted; so, Display 1 will turn on;
- when input 2 of the multiplexer is chosen, output 2 of the decoder is asserted; so, Display 2 will turn on;
- when input 3 of the multiplexer is chosen, output 3 of the decoder is asserted; so, Display 3 will turn on.

Therefore,

- when the output of the counter is 00, Display 0 will receive a $4F, hence displaying digit 3,
- when the output of the counter is 01, Display 1 will receive a $66, hence displaying digit 4,
- when the output of the counter is 10, Display 2 will receive a $6D, hence displaying digit 5,
- when the output of the counter is 11, Display 3 will receive a $7D, hence displaying digit 6.

Next, we will answer the following two questions to see if the circuit works as it is expected:

- How long does each number stay on display? In other words, how long does each display remain on?

- At what frequency does each display turn on?

Figure 34 shows a representative timing diagram for this multiplexing/demultiplexing system. The clock frequency is assumed to be 500 Hz, or the clock period is 2 ms. So, each display remains on for 2 ms every time it turns on. Additionally, each display turns on every 2 x 4 = 8 ms as there are four displays. In other words, each display turns on at the frequency of 500/4 = 125 Hz and stays on for 2 ms every time. As mentioned above, our eyes are unable to perceive flashing LEDs at 125 Hz. Therefore, the pattern 3456 will be seen on the four displays.

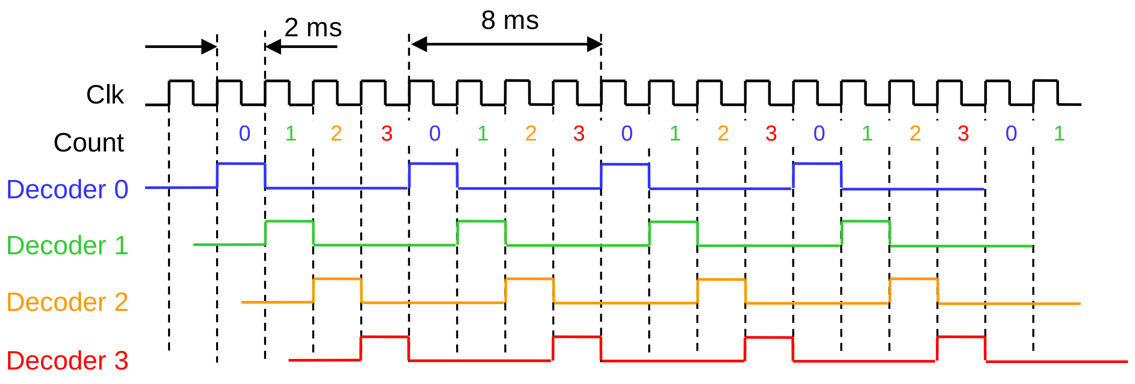


Figure 34. Timing diagram for the multiplexing/demultiplexing in Example 23

Multiplexing/Demultiplexing data on four 7-segment displays using the HSC12 microcontroller

In this section, we use the HCS12 microcontroller to display the same pattern (3456) on the four 7-segment displays of the Dragon 12+ trainer board. The hardware for this design is illustrated in Figure 35. Here is the software algorithm called *multiplexing/demultiplexing* to get the job done:

```
Do forever:
{
• Turn off all the displays
• Send the next digit to PORTB
• Turn on the next display
• Wait
}
```

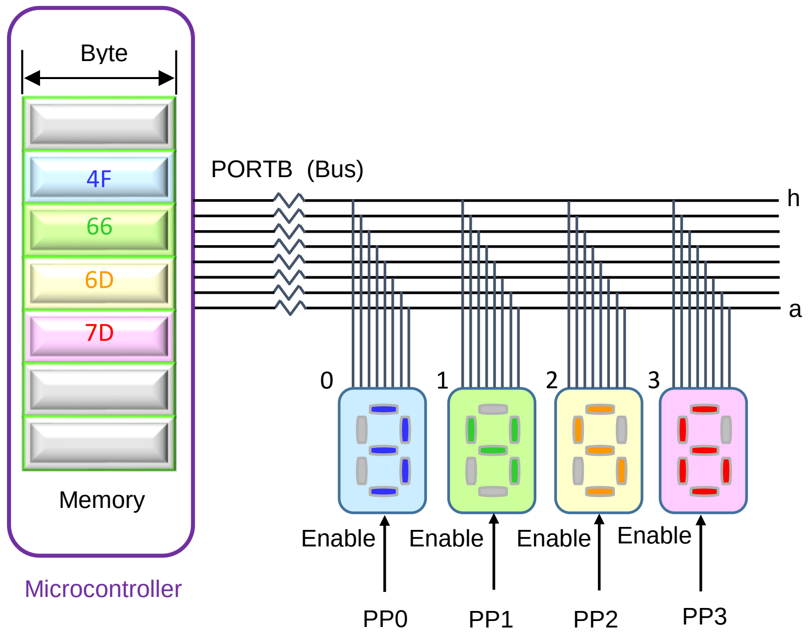


Figure 35. Hardware for software multiplexing/demultiplexing

Example 24. Using the above Multiplexing/Demultiplexing algorithm, display pattern 3456 on the 4 displays of the Dragon 12+ board. See Figure 35:

Do forever:

1.	PP0, PP1, PP2, and PP3 ← 0000	Turn off all displays
2.	PORTB ← $4F	Send digit 3 to PORTB
3.	PP0 ← 0, PP1, PP2, and PP3 ← 111	Turn on display 0 to display 3; leave the rest off
4.	Wait	Let the LEDs stabilize
5.	PP0, PP1, PP2, and PP3 ← 0000	Turn off all displays
6.	PORTB ← $66	Send digit 4 to PORTB
7.	PP1 ← 0, PP0, PP2, and PP3 ← 111	Turn on display 1 to display 4; leave the rest off
8.	Wait	Let the LEDs stabilize
9.	PP0, PP1, PP2, and PP3 ← 0000	Turn off all displays
10.	PORTB ← $6D	Send digit 5 to PORTB
11.	PP2 ← 0, PP0, PP1, and PP3 ← 111	Turn on display 2 to display 5; leave the rest off
12.	Wait	Let the LEDs stabilize
13.	PP0, PP1, PP2, and PP3 ← 0000	Turn off all displays
14.	PORTB ← $7D	Send digit 6 to PORTB
15.	PP23 ← 0, PP0, PP1, and PP2 ← 111	Turn on display 3 to display 6; leave the rest off
16.	Wait	Let the LEDs stabilize

In this example, the outputs of the PORTB register are the communication *bus* between the transmitters and the receivers; the data producers (transmitters) are the four memory locations from which the four digits are read and sent to the bus; the four displays are the consumers (or receivers). The multiplexing is performed in software, where one memory location at a time is read and sent to the bus. By turning on one display at a time while the correct data is on the bus, data is demultiplexed, i.e., the correct data is sent to the proper display.

Think critically:

- *Why did we turn off all the displays in each iteration in the above algorithm?*
- *Why did we not do so in Example 23?*

We are now going to translate the above algorithm into an assembly program. In the data segment, we create two arrays:

1. "decoder", which is a hexadecimal to 7-segment software decoder as shown here:

```
        org  $2000
decoder:
        dc.b $3F, $06, $5B, $4F, $66, $6D, $7D, $07, $7F, $6F, $77, $7C, $58, $5E, $79, $71
```

In this 16-byte array, the 7-segment codeword of the hexadecimal digit i, 0 ≤ i ≤ F, is seated in location i. This array is the software equivalent of the look-up table shown in Figure 27. For example, $3F, the 7-segment codeword of the hexadecimal digit 0, is sitting in location 0 of the array. As another example, the value placed in location 6 of the array is $7D, which is the 7-segment codeword of digit 6.

2. Four-byte array "buffer", which is initialized to 3, 4, 5, and 6 (or any four digits that you decide to display):

```
                org  $2500
buffer:         dc.b   3, 4, 5, 6
```

Here is the assembly code to get the job done. Review it rigorously. Note that the initialization stage is not shown here:

```
forever:
        ldy     #buffer         ; initialize buffer pointer
        ldab    #%11101110      ; Initialize register B, which drives the four common cathodes
        sec                     ; set carry bit; it is not affected by load/store/move
nextDigit:
        movb    #$F , PTP       ; Turn off all 7-segment displays
        movb    1, Y+, PORTB    ; read the next buffer location and send it to display
        stab    PTP             ; turn the next display on
        jsr     wait3m          ; wait for 3 ms, subroutine wait3m is shown after this main program
        rolb                    ; get ready for the next digit
        bcs     nextDigit       ; move on to the next digit, if any
        bra     forever
```

Take a second look at the following instruction, which writes register B into PTP:

```
stab PTP
```

We know that the four LSbs of PTP drive the four common cathodes; on the other hand, instruction {ldab #%11101110} initializes register B to binary 1110111**0** before the loop nextDigit begins. Therefore, in the first iteration of this loop, Display 0 will turn on (because the LSb of 1110111**0** is **0**). Then instruction {rolb} rotates Register B to the left changing its contents to 110111**0**1. When this pattern is seated in PTP, Display 1 will turn on, and so on. Note that instruction {sec} sets the carry flag to 1 before the loop nextDigit begins as a 0 in the carry flag will erroneously change the content of Register B to 11011100 when instruction{rolb} executes for the first time, and this value will turn two displays on (instead of one).

Register B is a dual-purpose register: in addition to providing the correct 4 LSbs for PTP, Register B is also used as a loop counter: Look at the second **0** in bit position 4 of 111**0**1110, the initial value of Register B. After four rotations carried out by instruction {rolb} in four iterations, and when all the four digits sitting in the array "buffer" have been displayed, this **0** reaches the carry flag, and then instruction {bcs} detects this **0** and ends the nextDigit loop.

Subroutine wait3m is shown here:

```
wait3m:     pshx
            pshc
            ldx      #24000    ; initialize loop counter (an 8-bit register is not enough)
   wait:    dbne     X, wait   ; wait here for 24000 x 3 E-cycles or 3 ms
            pulc
            pulx
            rts
```