COE 306, Term 161

Introduction to Embedded Systems

**Assignment# 2 Solution**

**Due date: Saturday, Oct. 22, 2016**

# A system has two memory-mapped I/O devices. The first device has an 8-bit status register at address 0xA0, immediately followed by a 32-bit data register. The second device has a 16-bit status register at address 0xB0, followed by a 32-bit data register. The first device is used to receive data (i.e., input device). The least-significant bit in the status register is a *data ready flag*, which is set automatically by the device whenever new data are received. For the device to receive more data, the *data ready flag* must be manually reset by software to indicate that the current data have been processed.

# The second device is used to send data (i.e., output device). The least-significant bit of its status register is a read-only *ready to send flag*, and the second least-significant bit (bit 1) is a *transmit enable* command bit that is automatically reset by the device after each transmission.

# We would like to write software that collects 32-bit words received through the first device, and computes the average of received data until the second device becomes ready to send. Once the second device becomes ready to send data, the average word is sent using the second device. Once the average is sent, the average computation is restarted for the next sample of data, ignoring the previously received data samples.

* 1. Write a C program that implements this behavior using polling only.

#define DEV1\_STATUS 0xA0

#define DEV1\_DATA 0xA1

#define DEV2\_STATUS 0xB0

#define DEV2\_DATA 0xB2

int main(void) {

unsigned int sum = 0; // holds sum of data

unsigned int count = 0; // holds count of data

while(1) {

if ((\* (char \*) DEV1\_STATUS) & 1) { // data ready flag is set

sum += (\* (int \*) DEV1\_DATA);

count++;

(\* (char \*) DEV1\_STATUS) &= 0xfe; // reset data ready flag

}

if ((\* (char \*) DEV2\_STATUS) & 1) { // ready to send

(\* (int \*) DEV2\_DATA) = sum/count;

(\* (char \*) DEV2\_STATUS) |= 2; // transmit enable

sum = 0;

count=0;

}

}

}

* 1. Assuming that each device has its own interrupt handler, write the handlers for each device in C. The first device generates an interrupt request upon receiving new data. The second device generates an interrupt request upon becoming ready to send new data.

Use the signatures:

**void** device1\_handler(**void**);

**void** device2\_handler(**void**);

unsigned int sum = 0;

unsigned int count = 0;

void device1\_handler(void) {

sum += (\* (int \*) DEV1\_DATA);

count++;

(\* (char \*) DEV1\_STATUS) &= 0xfe; // reset data ready flag

}

void device2\_handler(void) {

(\* (int \*) DEV2\_DATA) = sum/count;

(\* (char \*) DEV2\_STATUS) |= 2; // transmit enable

sum = 0; count=0;

}

1. Write a program that implements a decimal up-counter that counts from 0 to 9 and back to 0 (i.e., 0→1→2→…→9→0) and displays the result in a seven-segment display. Check the seven-segment display part number you have and then find out its data sheet to identify the configuration of pints. The MAN72A-like seven-segment display is shown below along with its pin configurations:

|  |  |
| --- | --- |
|  |  |

# 

1. Add a switch to your implementation and configure it as an input and use it based on input polling such that whenever the switch is pressed the direction of the counter changes i.e., from up counting to down counting or vice versa.

**int** **main**(**void**) {

LPC\_GPIO0->FIODIR = (1<<11);

LPC\_GPIO0->FIOPIN = (1<<11);

LPC\_GPIO2->FIODIR = 0x7f;

LPC\_GPIO0->FIODIR &= ~(1<<9);

**int** count;

**int** i, x, Dir=0;

**while**(1) {

x = LPC\_GPIO0->FIOPIN & (1<<9);

x = x >>9;

**if** (!x) {Dir = 1-Dir; }

LPC\_GPIO2->FIOPIN = ~decode(count);

**for**(i = 0; i < 5000000; i++);

**if** (!Dir) {

count=count+1;

**if** (count>9) count=0;

}

**else** {

count=count-1;

**if** (count<0) count=9;

}

}

**return** 0 ;

}

**int** **decode**(**int** num){

**int** res; // GFEDCBA

**if**(num == 0)

res = 0x3F;

**else** **if**(num == 1)

res = 0x6;

**else** **if**(num == 2)

res = 0x5B;

**else** **if**(num == 3)

res = 0x4F;

**else** **if**(num == 4)

res = 0x66;

**else** **if**(num == 5)

res = 0x6D;

**else** **if**(num == 6)

res = 0x7D;

**else** **if**(num == 7)

res = 0x7;

**else** **if**(num == 8)

res = 0x7F;

**else**

res = 0x6F;

**return** res;

}

1. Use the switch in the same way as used in part (b) to change the direction of counting but based on the use of interrupt instead of using polling. Write an input interrupt handler for the switch such that whenever the switch is pressed the handler changes the value of a global variable that changes the direction of counting.

**int** Dir=0;;

**void** **EINT3\_IRQHandler**()

{

Dir = 1-Dir;

LPC\_GPIOINT->IO0IntClr |= 1<<9;

}

**int** **main**(**void**) {

LPC\_GPIOINT->IO0IntEnR |= 1<<9;

NVIC\_EnableIRQ(21);

LPC\_GPIO0->FIODIR = (1<<11);

LPC\_GPIO0->FIOPIN = (1<<11);

LPC\_GPIO2->FIODIR = 0x7f;

**int** count=0;

**int** i;

**while**(1) {

LPC\_GPIO2->FIOPIN = ~decode(count);

**for**(i = 0; i < 5000000; i++);

**if** (!Dir) {

count=count+1;

**if** (count>9) count=0;

}

**else** {

count=count-1;

**if** (count<0) count=9;

}

}

**return** 0 ;

}

**int** **decode**(**int** num){

**int** res; // GFEDCBA

**if**(num == 0)

res = 0x3F;

**else** **if**(num == 1)

res = 0x6;

**else** **if**(num == 2)

res = 0x5B;

**else** **if**(num == 3)

res = 0x4F;

**else** **if**(num == 4)

res = 0x66;

**else** **if**(num == 5)

res = 0x6D;

**else** **if**(num == 6)

res = 0x7D;

**else** **if**(num == 7)

res = 0x7;

**else** **if**(num == 8)

res = 0x7F;

**else**

res = 0x6F;

**return** res;

}

1. Discuss your implementations in part (b) and (c) and which implementation is better and why?

When polling is used we can observe that sometimes when the switch is pressed it is not captured as this could be when the switch is pressed while CPU is executing the code waiting for the given delay or performing the display. On the other hand, based on using interrupt whenever the switch is pressed the interrupt handler code is executed which changes count direction. Thus, no switch press is missed.

1. Include a link for a video demo for your implementations of part (c) and (d) on the LPCXpresso board.