1)

1/ Circular Buffer and UART

2

while (1) {

algorithm();

IEC1bits.U2RXIE = 0;

int read = cb\_pop(&cb, &readChar);

IEC1bits.U2RXIE = 1;

if (read == 1) {

[...]

}

tmr\_wait\_period(TIMER1);

}

CORRECT ANSWER:

The loop reads from the CB and consumes one byte. Therefore, it is not correct.

PROBLEM:

In 10ms I read 1 char per ms, thus, if I have 10 characters, I read only one of them and leave the others in the buffer.

After some iterations the CB becomes full and thus I loose some data.

SOLUTION:

Read more than 1 char in a loop. Also make the buffer size long enough.

2)

2/ UART

void uartWrite(){

while(!U1STAbits.TRMT);

char temp[5];

sprintf(temp, "%d", charRecvNum);

for(int i=0;i<strlen(temp);i++){

char c = temp[i];

if(c !='\0')

U2TXREG = c;

}

}

CORRECT ANSWER:

The code correctly waits until previous transmissions are completed, however, if the data to be sent is longer than 4 bytes, then FIFO overflow would occur.

OR

The code correctly waits until previous transmissions are completed, Data is assumed to be shorter than 5 bytes, hence space is assured in the FIFO. However, the practice is unsafe.

PROBLEM:

It is clear

SOLUTION:

Always wait for data?

3)

3/ Shared variables handling

4

void \_U2RXInterrupt(){

IFS1bits.U2RXIF = 0;

buf.cbuf[buf.write\_p] = U2RXREG;

buf.write\_p ++;

buf.chars2read ++;

if(buf.write\_p == BUFFER\_SIZE){

buf.write\_p = 0;

}

}

int main() {

...

while(1) { algorithm();

while (buf.chars2read > 0){

[...]

buf.read\_p ++;

buf.chars2read --;

if(buf.read\_p == BUFFER\_SIZE){

buf.read\_p = 0;

}

count++;

}

}

CORRECT ANSWER:

If the Interrupt happens during the buf.chars2read --;, then the variable will be incremented in the ISR, but the initial value will be decremented. The final value will be the initial minus one, which is incorrect.

OR

If the buf.chars2read--; is atomic there is no problem.

PROBLEM:

You have to guarantee that the operation is atomic. Otherwise, if you don't know if the op is atomic, you have to protect this operation.

The chars2read is loaded in CPU, however if an interr occurs, it will modify the value in memory, that will not be up to date.

SOLUTION:

Disable interr before doing the chars2read-- and restore it immediately after.

4)

4/ What's wrong with the previous

interrupts? Why the debouncing will not

work?

5

void \_INT0Interrupt(){

IFS0bits.INT0IF = 0;

IEC0bits.INT0IE = 0;

T3CONbits.TON = 1;

}

void \_T3Interrupt(){

IFS0bits.T3IF = 0;

[...]

IEC0bits.INT0IE = 1;

IEC1bits.INT1IE = 1;

IFS0bits.INT0IF = 0;

IFS1bits.INT1IF = 0;

}

First clear the flag, then re enable the interrupt.

5)

5/ Interrupts and main

6

void \_INT1Interrupt(){

IFS1bits.INT1IF = 0;

clearFirstRowLCD();

clearSecondRowLCD();

sprintf(str,"%d ",charCounter);

writeStringLCD(str, 5);

[...]

}

int main () {

[...]

while(1) {

int read = readBufferFromUART(&cb, &val);

if (read == 1) {

[...]

moveCursorLCD(SECOND\_ROW\_INDEX, 0);

writeStringLCD("Char Recv:", 10);

moveCursorLCD(SECOND\_ROW\_INDEX, 11);

sprintf(str, "%d", charCounter);

writeStringLCD(str, 5);

previousValue = currentValue;

}

[...]

}

}

CORRECT ANSWER:

Code is incorrect.

PROBLEM:

Processing time too long to be executed inside an interrupt.

Also using the variables both in the main and in the interrupt requires a safty mechanism to handle the shared peripheral.

SOLUTION:

Disable interrupt, do stuff, then re-enable it. But you'll have significant delays.

OR

Rewrite the logic and move the code out of the interrupt. Thus removing access to the SPI in the interrupt.

6)

6/ Interrupts coding

7

void \_T2Interrupt() {

IFS0bits.T2IF = 0;

if (!btn\_press)

pinValue = PORTEbits.RE8;

else

pinValue = PORTDbits.RD0;

T2CONbits.TON = 0;

if (pinValue) {

if (!btn\_press) {

char buff[LINE\_SIZE];

sprintf(buff, "%d", charCount);

for (int i = 0; i < strlen(buff); i++) {

while (!U2STAbits.TRMT);

U2TXREG = buff[i];

}

}

}

}

CORRECT ANSWER:

The interrupt correctly clears the IF, and proceeds with sending data over to the UART. However, using busy waiting means that it will be blocked for 1 ms for each character if the FIFO buffer is full.

OR

Using a busy waiting paradigm inside the ISR is not recommended and therefore the code should be considered as wrong.

PROBLEM:

busy waiting implies that the interrupt is long.

UXTF != TRMT, TRMT implies a delay every time I send on the channel. UXTF doesn't delay for each character.

SOLUTION:

Use always UXTF and don't do stuff in interrupts.

7)

7/ What is wrong with the debouncing

solution?

8

Void \_INT0Interrupt(){

IFS0bits.INT0IF = 0;

IEC0bits.INT0IE = 0;

IEC0bits.T1IE = 1;

IFS0bits.T1IF = 0;

TMR1 = 0;

T1CONbits.TON = 1;

}

Void \_T1Interrupt(){

IFS0bits.T1IF = 0;

IEC0bits.T1IE = 0;

IEC0bits.INT0IE = 1;

if(PORTEbits.RE8 == 1){

flags\_interrupts = 1;

}

}

CORRECT ANSWER:

PROBLEM:

SOLUTION:

8)

8/ Debouncing

void \_INT0Interrupt() {

IFS0bits.INT0IF = 0;

IFS0bits.T3IF = 0;

IEC0bits.T3IE = 1;

if (intr0\_flag == 1) {

intr0\_flag = 0;

T3CONbits.TON = 1;

tx\_flag = 1;

}

}

void \_T3Interrupt() {

IFS0bits.T3IF = 0;

intr0\_flag = 1;

}

CORRECT ANSWER:

The solution is incorrect because it does not check the state of the button when the timer elapses. Furthermore, T3 will keep on expiring because it is not stopped.

PROBLEM:

The timer is enabled and let always on and the enable from the interrupt is never removed

SOLUTION: