

Embedded Systems Exam

Name: Ahmed ElShaarany

Date: 7/8/2015

Problems

P1: (40 points) Debouncing button press ripples

Maintain and provide statistics with the following information:

- Total number of LED toggles that were performed (Did this equal the number of times you have pressed the button?)

Yes, pressing the Push Button 10 times results in 10 LED toggles.

- Total number of spurious button press interrupts that were chosen not to be serviced

By watching the expression that counts the number of spurious button presses, I was able to see that most of the time the number was 0. The maximum number was 6. Other times, I was getting numbers like 2, 3, and 4.

- The maximum observed ripple duration.

By watching the expression that records the timer value at the last spurious button press, I was able to see that most of the time the number was 0. The frequency of operation was 128000 Hz. The maximum was 76 counts (0.5 msec). Other times, I was getting numbers like 4, 10, 12, ... counts at a clock frequency of 128000 Hz.

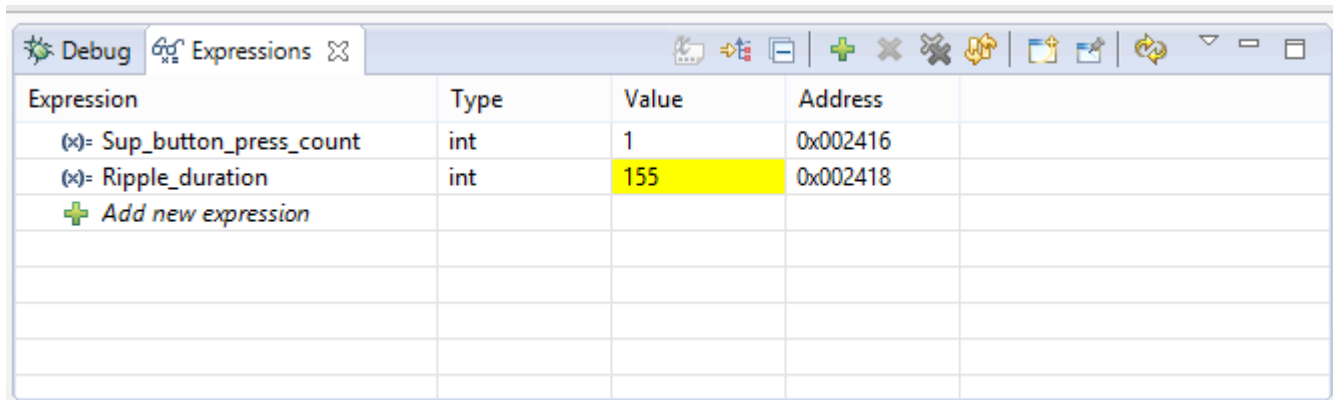
- Explain the main idea behind your approach

The main idea behind my approach was as follows. Once the actual push button is received, the ISR for the button press would toggle the LED and trigger a timer. During the duration of that timer, if any push buttons were received, these would be considered as spurious button presses and would not toggle the LED. After the timer is done, the timer ISR would then stop the timer and then enable again the ability to push the button. This was done using a global variable that would enable or disable button presses.

The timer duration had to be small enough that it would not block actual consecutive button presses (i.e. if the push button is pressed at a high frequency, it would still toggle the LED).

The parameter that was tuned in this algorithm was the duration of the timer. In order to measure the total number of spurious button presses and ripple duration, I had to maximize the duration of the timer. The intention was not to implement the actual algorithm using this maximum timer duration, but it was merely to make sure that I would not miss any spurious presses and to calculate the ripple duration correctly. The maximum timer duration was 0.5 sec (8.192MHz frequency with a 64 divider value). After calculating the maximum ripple duration (0.5 msec), I found that it was safe to use a timer duration of 8 msec (8.192MHz with 1 divider value). This value would ensure that all ripples have passed and would be very adequate in case of very fast successive actual button presses. The 8 msec timer duration would allow a maximum of 125 actual button presses per second which is extremely fast (i.e. much faster than one can successively press the push button).

An example of the results:



The screenshot shows a debugger's 'Expressions' window. It contains a table with four columns: 'Expression', 'Type', 'Value', and 'Address'. The first row shows 'Sup_button_press_count' with a value of 1. The second row shows 'Ripple_duration' with a value of 155, which is highlighted in yellow. Below these is a button to 'Add new expression'.

Expression	Type	Value	Address
(x)= Sup_button_press_count	int	1	0x002416
(x)= Ripple_duration	int	155	0x002418
+ Add new expression			

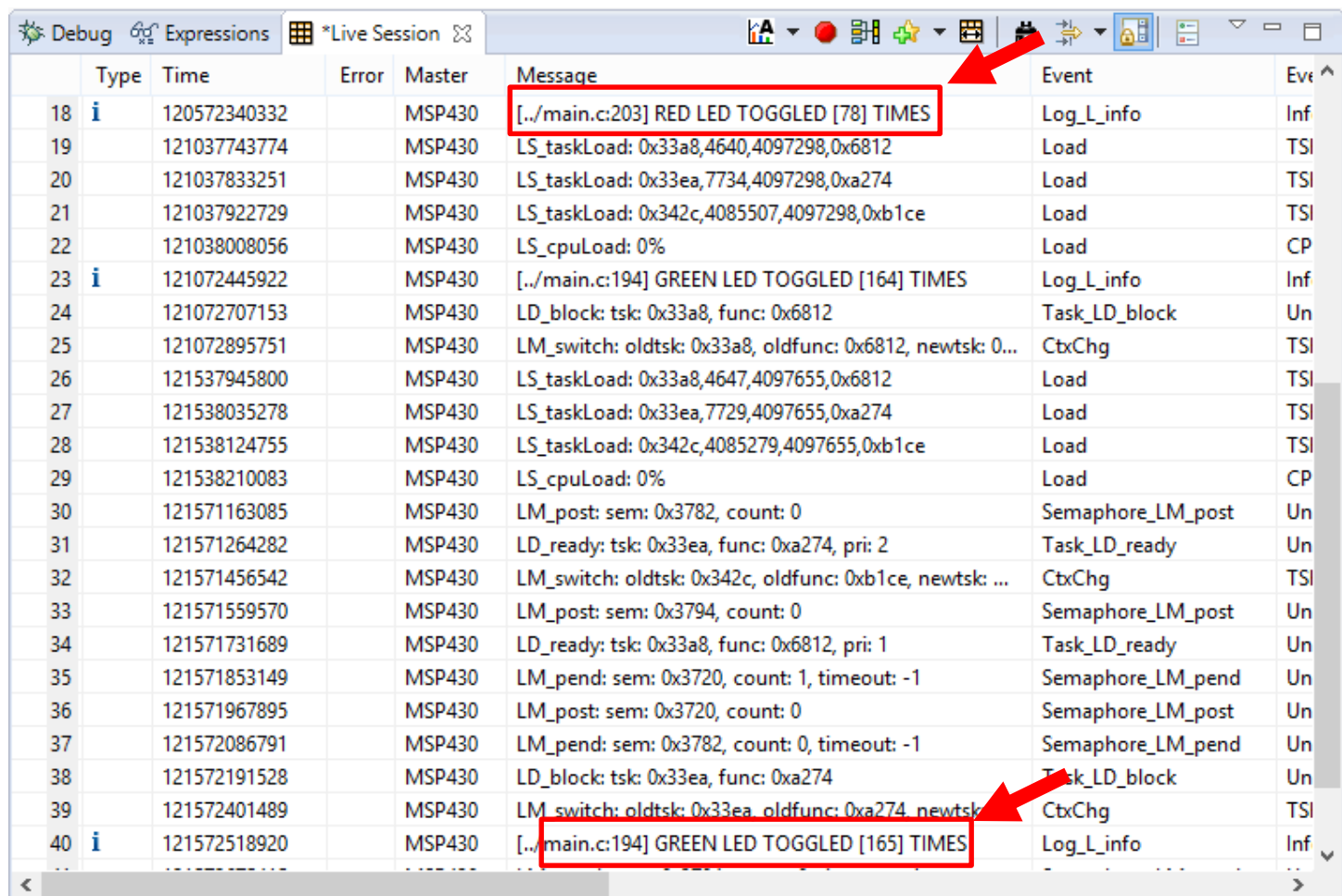
From the figure above, we can see that we encountered 1 spurious button press and the maximum ripple duration was 155 counts at a frequency of 8.192 MHz which means that it was 0.02 msec which is much less than the chosen maximum timer duration which is 8 msec.

P2: (30 points) Random selection of LED to toggle

- Count toggles for GREEN and RED LED's. Provide counts for a 2 minute run.

In an ideal situation, in 2 minutes with 0.5 sec toggles, the number of GREEN LED toggles would be 160 and the number of RED LED toggles would be 80 with a total of 240 toggles. The actual numbers I got from one run of the program was 165 GREEN LED toggles and 78 RED LED toggles with a total of 243 toggles. These numbers would provide a very close value to the desired probability. For the GREEN LED, the desired probability was 0.67 (i.e. 2/3) and the actual was 0.68 (i.e. 165/243). For the RED LED, the desired probability was 0.33 (i.e. 1/3) and the actual was 0.32 (i.e. 78/243).

The numbers from the live session logging are shown in the below figure.



Type	Time	Error	Master	Message	Event	Ev
i	120572340332		MSP430	[./main.c:203] RED LED TOGGLED [78] TIMES	Log_L_info	Inf
	121037743774		MSP430	LS_taskLoad: 0x33a8,4640,4097298,0x6812	Load	TSI
	121037833251		MSP430	LS_taskLoad: 0x33ea,7734,4097298,0xa274	Load	TSI
	121037922729		MSP430	LS_taskLoad: 0x342c,4085507,4097298,0xb1ce	Load	TSI
	121038008056		MSP430	LS_cpuLoad: 0%	Load	CP
i	121072445922		MSP430	[./main.c:194] GREEN LED TOGGLED [164] TIMES	Log_L_info	Inf
	121072707153		MSP430	LD_block: tsk: 0x33a8, func: 0x6812	Task_LD_block	Un
	121072895751		MSP430	LM_switch: oldtsk: 0x33a8, oldfunc: 0x6812, newtsk: 0...	CtxChg	TSI
	121537945800		MSP430	LS_taskLoad: 0x33a8,4647,4097655,0x6812	Load	TSI
	121538035278		MSP430	LS_taskLoad: 0x33ea,7729,4097655,0xa274	Load	TSI
	121538124755		MSP430	LS_taskLoad: 0x342c,4085279,4097655,0xb1ce	Load	TSI
	121538210083		MSP430	LS_cpuLoad: 0%	Load	CP
	121571163085		MSP430	LM_post: sem: 0x3782, count: 0	Semaphore_LM_post	Un
	121571264282		MSP430	LD_ready: tsk: 0x33ea, func: 0xa274, pri: 2	Task_LD_ready	Un
	121571456542		MSP430	LM_switch: oldtsk: 0x342c, oldfunc: 0xb1ce, newtsk: ...	CtxChg	TSI
	121571559570		MSP430	LM_post: sem: 0x3794, count: 0	Semaphore_LM_post	Un
	121571731689		MSP430	LD_ready: tsk: 0x33a8, func: 0x6812, pri: 1	Task_LD_ready	Un
	121571853149		MSP430	LM_pend: sem: 0x3720, count: 1, timeout: -1	Semaphore_LM_pend	Un
	121571967895		MSP430	LM_post: sem: 0x3720, count: 0	Semaphore_LM_post	Un
	121572086791		MSP430	LM_pend: sem: 0x3782, count: 0, timeout: -1	Semaphore_LM_pend	Un
	121572191528		MSP430	LD_block: tsk: 0x33ea, func: 0xa274	Task_LD_block	Un
	121572401489		MSP430	LM_switch: oldtsk: 0x33ea, oldfunc: 0xa274, newtsk: ...	CtxChg	TSI
i	121572518920		MSP430	[./main.c:194] GREEN LED TOGGLED [165] TIMES	Log_L_info	Inf

Extra credit: (20 points) Clock functions (SWI's) and Rate Monotonic priorities

Answer the following questions:

- Which priorities did you assign to each of the clock function SWI's?

Since we are using the Rate Monotonic priority assignment, so the shorter the duration of the task, the higher the task's priority. So the priorities for the three clock function SWI's were as follows:

1. Period 500ms, duration ~10ms, toggles red LED (Highest Priority: 3)
2. Period 200ms, duration ~30ms, toggles green LED (Middle Priority: 2)
3. Period 1s, duration ~600ms, toggles *blocking* of green LED toggling (Lowest Priority: 1)

- What are the values of input argument to the delay function within each of the clock function SWI's?

RED LED toggle delay function input argument: 80000

GREEN LED toggle delay function input argument: 240000

GREEN LED BLOCK toggle delay function input argument: 4800000

- How did you calculate the values for the delay function?

The clock frequency used is 8000000 Hz. Therefore, for calculation of the input arguments:

RED LED toggle: $8000000 \times 10 \text{ msec} = 80000 \text{ cycles}$

GREEN LED toggle: $8000000 \times 30 \text{ msec} = 240000 \text{ cycles}$

GREEN LED BLOCK toggle: $8000000 \times 600 \text{ msec} = 4800000 \text{ cycles}$

Questions

Q1: (8 points)

State at least four typical types of peripherals found in microcontrollers.

1. GPIO
2. Timers
3. Clocks
4. Power Management
5. Communications [Serial ports, USB, Radio]
6. Hardware Accelerators

Q2: (12 points)

State at least four reasons for using modularization or 'divide and conquer' approach when developing embedded software.

1. The 'divide and conquer' approach produces programs which are easy to manage, understand and test.
2. Program development can be done on an incremental basis ('a bit at a time').
3. Errors are easier to track down and correct.
4. Modifications are usually localized, so maintaining high program stability.
5. Portability can be increased by burying machine-specific features within designated modules. Thus, when moving to other target systems, only these modules need changing.
6. Libraries of useful functions can be built up (especially true of complex maths functions), so leading to software re-use.
7. It is easier to attack the problems of slow code and memory inefficiency when using modular construction (the 'code optimization' requirement).
8. In a large system, development times can be pruned by using a team of designers working in parallel. Although various work-sharing methods could be used, allocating work on a module basis has two particular advantages. First, it enables programmers to work independently; second, it simplifies integration of the resulting code.

Q3: (10 points)

What are the RTOS timers typically used for? State at least two examples.

In embedded systems, scheduling system and user tasks, to run for/after a specified duration, is required. Therefore, in order to keep track of time delays and timeout, we need a module that triggers periodic interrupts. This module is the RTOS timer.

Examples:

- Toggle an LED periodically using the duration of the timer as the period.
- Schedule toggling different LED's with different periods of time.
- Schedule an LED to toggle only once using the one-shot option of the RTOS timer