Name: Christopher Shu Chun Lam

Created: 17/11/2019
Last modified: 23/11/2019

COMP9032 2019 T3 Project – Design Manual

Table of Contents

Name: Christopher Shu Chun Lam

Created: 17/11/2019
Last modified: 23/11/2019

Project Description:

The program simulates control operations in an airplane to increase/decrease the opaqueness of passenger smart windows with a lab board provided by UNSW.

NOTE: Request is used to refer to a command to change window level(s) from either of 3 controls

Project Specifications/Requirements:

- Four windows with four opaque levels (level 0-3) represented by brightness on LEDs. 2 LED bars for each window. The brighter the LEDs, the darker the window.
- Three controlling situations in decreasing priority:
  - Emergency control: All windows set immediately to clear(0)
  - Central control: Set all windows to clear (0) or to dark(3)
  - Local (Individual) control: Increase/decrease window level by 1
- 0.5 second delay to change a window opaque level to another level
- Delays are not shared between windows
- Delays are not shared between requests for the same window (from forum discussion)
- Local controls can be done in parallel

Input:

- Push button for emergency control
- 2 keys on keypad for increase to dark/decrease to clear in central control
- 2 keys on keypad for each window for increase/decrease by 1 level in local control

Output (other than LED as specified above):

- LCD provides textual information about simulation. Left of LCD shows state of simulation, right of LCD shows opaque levels of each window

Implementation

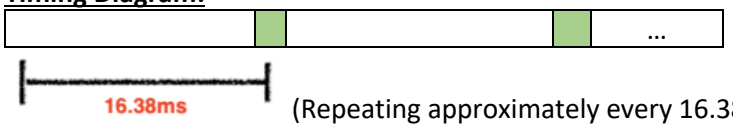
Program timing (centralTimer interrupt)

Timer0 OVF0 is used to implement a single central Timer (centralTimer interrupt) with pre-scaling of /1024

- Interrupt executes every 256*1024 = 262,144 clock cycles
- Lab board clock speed=16Mhz

Therefore, centralTimer interrupt raised every 262,144/16,000,000 ≈ 16.384ms
Therefore, ≈ 30.52 (to 2.d.p) interrupts occur between every 0.5sec delay

Timing Diagram:



(Repeating approximately every 16.38ms)

	Task1: Keypad polling and emergency interrupt (when called)
	Task2: CentralTimer interrupts

Generating delay (timeCount)

To generate a delay, a timeCount register is used which was incremented for each centralTimer interrupt raised (as a form of clock). By adding +31 to the value in timerCount when a request is raised, it can be determined when a request is to be executed to replicate a 0.5sec delay.

value to execute request = timeCount when request raised + 31

Name: Christopher Shu Chun Lam **Created**: 17/11/2019
Last modified: 23/11/2019

Input: Keypad and Push Button

Keypad and debouncing:

Parallel Input / Polling

Keypad uses polling method of outputting 1 column of keypad input to 0 at a time and reading input from that column. To allow parallel input, the entire column is treated as a single input, which when combined with debouncing described below, allows input from multiple buttons to be processed at the same time.

Input from rows is cleaned using a mask (rowMask) to isolate input bits of the PIN (since the other 4 of them are configured for output). LOGICAL AND combines the row inputs with request register (described later). Finally, it is modified with control bits to determine type of request before incrementing to the next col (if central control request is not raised. Else, keypad only polls the leftmost column)

Debouncing

A counter passed approach to debouncing is used. When reading in input from each column of the keypad, the counter is initialised to 15. Every 1ms, input is read from the keypad.

- If no input is detected, decrease counter by 1
- Else, use LOGICAL AND of input with request register to copy any pressed bits (0) to request register

Once counter reaches 0, it is considered stable and debouncing ends.

Push Button

Push button set to falling edge interrupt and raises emergency interrupt when push button (PB0) is pressed. All window levels are immediately set to clear (0), queues are cleared (queueHead=queueTail. Described in data structure section later) and interface is updated.

Control priority:

- If push button for emergency is pressed, register emergencyState is set to 1 and queues are emptied (described later). This register prevents any partial polling from being executed and is cleared once polling is reset to initial state
- If a request from central control is requested, queues are emptied and the number of central control requests in queue (centralState) is incremented. While centralState != 0, local requests cannot be pushed onto queue data structure.

Output: Interface

LED brightness for every window and LCD display is updated after any type of request is processed (including from emergency control).

LED brightness:

- LEDs use phase correct PWM using compare registers with 1 for each window:
  - Timer4 compare register C (Window1)
  - Timer5 compare registers A,B,C (Window4,3,2)

To implement LED lights that can represent 4 levels of brightness, pulse width modulation using timer overflow compare registers were used. Given the ports on the lab board used for other purposes, pins for timer4 and timer5 were available and used. By using standard settings for these timers in phase correct PWM with a top of 0xFF, only the low-byte register of the 2-byte compare registers were required to generate brightness of 4 different levels.

The four opaqueness levels are defined in directives as PWMclear, PWMlight, PWMmedium, and PWMdark. To modify brightness of LEDs for each window to their respective levels, simply output PWMvalue to the low-byte compare register for the desired window.

LCD:

LCD is initialised to default configuration and display as defined in project specifications. Since only 5 sections of LCD were required (1 for control state and 4 for the windows) to be modified during program runtime, the instructions to set current data memory address on LCD to the desired section were stored as directives as top of file.

To modify textual information on LCD, run do_lcd_command macro to set address to desired section of LCD and run do_lcd_data or do_lcd_data_reg to modify data in screen with ASCII value stored in desired register.

Data Structures: Storing and processing requests from control states

Window levels

Window levels stored as 1-byte variable (in data memory) labelled as win1_level, win2_level, win3_level, win4_level. They are used by operations in program to set window levels.

Requests (Request Register)

- Requests for state changes from central/local control are converted from row input of keypad and interpreted using control bits

X = Unused bit
N = Valid bit for request
central_bit = specifies request from central control
localInc_bit = specifies request from local control

Central control requests (Bit 7-0)

X	Central_bit	localInc_bit	X	Win1	Win2	Win3/centralInc	Win4 /centralDec
X	1	X	X	X	X	N	N

- If both centralInc/centralDec set/clear, do nothing. Invalid request
- If only centralInc clear, request is for all windows to be set to window level 3
- If only centralDec clear, request is for all windows to be set to window level 0

Local control requests (Bit 7-0)

X	Central_bit	localInc_bit	X	Win1	Win2	Win3/centralInc	Win4 /centralDec
X	0	N	X	N	N	N	N

- If localInc_bit set, request is for window level increases, else request is for window level decreases
- For win1-4, if bit clear, request is applied to the respective windows. Else, for each window, do nothing for a window if bit set

Queues (reqQueue, timeQueue):

- 2 parallel array-based circular queue data structures used (in data memory)
  reqQueue: For queueing requests

timeQueue: For queueing time to execute requests at same node position in reqQueue

Since requests from central and local control require a 0.5sec delay before implementing their changes, and requests for each window have their own delay, the only limitations on the number of requests waiting to be executed is solely dependent on hardware limitations, human limitations (e.g. button press speed), and software limitations due to debouncing implementation. Therefore, a queue was implemented in data memory as it allowed delaying a variable number of requests in comparison to other fixed approaches.

Queue push/pop operations

- Used for storing and retrieving requests from queues like C array based queues

To push onto queue, INT1 bit is enabled to activate pushQueue interrupt (enablePushQueue), pushing data in request register to queueTail of reqQueue. To pop off queue, popReqQueue macro retries data at queueHead of reqQueue and storesit in request register.

address of desired node in queue = Queue label address in dseg + displacement

For the displacement value, two variables (registers) were used: queueHead & queueTail. These variables tracked the displacement required to access the head and tail nodes of both queues.

- queueHead & queueTail work in parallel for both queues (e.g. incrementing queueHead after pop increments head of both queues)
- queueHead incremented for each node popped off reqQueue
- queueTail incremented for each request pushed onto reqQueue
- New requests were pushed to the end of queue and oldest requests popped from front of the queue. When queueHead==queueTail, queue is empty.

To ensure the array-based queue did not overflow, a circular queue was imitated to allow reuse of previously popped nodes. By setting a queueSize of 256, displacement variables queueHead and queueTail loop from 255 to 0 when incremented as long as effects on SREG are ignored.

Storing requests

When a request is registered in keypad polling, a mask (rowMask) is first applied to clean the request of irrelevant data. The request is converted dependent on column number to set control bits and pushed onto reqQueue at the node of position reqQueue + queueTail. Another request (timeCount + 31) is pushed onto the same tail node position of timeQueue as explained in "Generating delay" above.

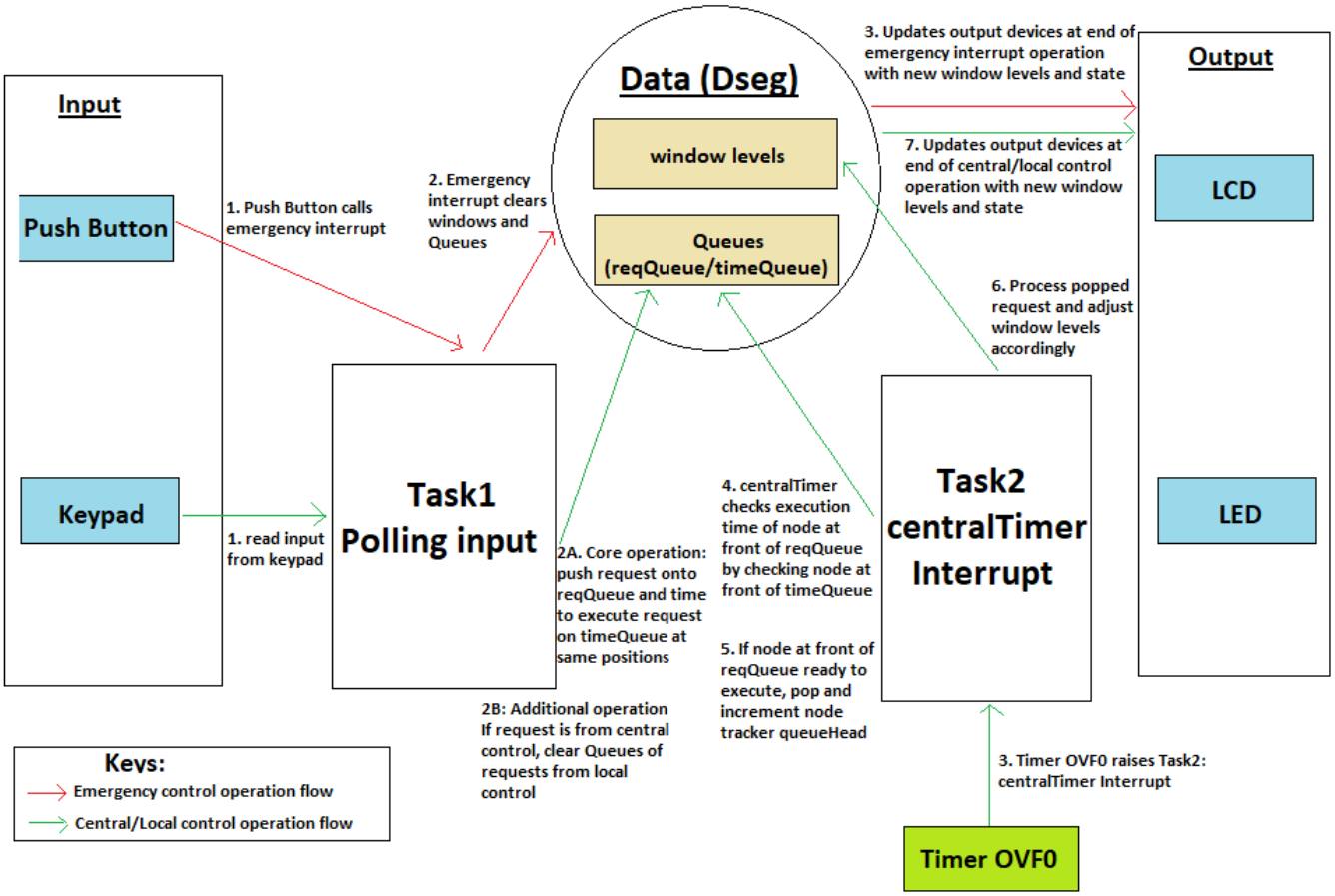
Processing requests

When centralTimer interrupt is raised, if the queues are not empty (queueHead != queueTail), check the value stored in head node of timeQueue (timeQueue + queueHead) and compare with current timeCount value to determine if the oldest queued request is ready to be executed. If so, the request is popped from the same position of reqQueue and processed depending on control bits to determine the type of request and which windows levels are to be modified.

Name: Christopher Shu Chun Lam

Created: 17/11/2019
Last modified: 23/11/2019

Block diagram of program flow:



Operating procedure:

(Based on UNSW provided lab board configuration)

Operating keypad/push button:

Keypad:

1 (Unused)	**2** Local Control Inc window1 level	**3** Local Control Dec window1 level	**A** (Unused)
4 (Unused)	**5** Local Control Inc window2 level	**6** Local Control Dec window2 level	**B** (Unused)
7 Central Control Increase to dark	**8** Local Control Inc window3 level	**9** Local Control Dec window3 level	**C** (Unused)
*** Central Control Decrease to clear	**0** Local Control Inc window4 level	**=** Local Control Dec window4 level	**D** (Unused)

Push Button

Push Button (PB0)	Push button raises emergency interrupt request to set all window levels to clear(0) immediately

Name: Christopher Shu Chun Lam

Created: 17/11/2019
Last modified: 23/11/2019

Wiring / Setup:

Component	I/O direction	I/O pins	AVR pins
LED	Output	LED0 - LED1	PH8
		LED2 - LED3	PL2
		LED4 - LED5	PL3
		LED6 - LED7	PL4
LCD Data	Input / Output	D0 - D7	PC7 – PC0
LCD CTRL	Output	BE – RS	PA4 – PA7
Push Button	Input	PB0	RDX4
Keypad	Input	R3 – R0	PF0 – PF3
	Output	C0 – C3	PF4 – PF7

Additional comments on operating procedures:

- Pressing buttons in a state with higher priority will cause requests from lower priority states to not execute (as described in specification on priority of control states)
- Pressing both increase/decrease for central control is ignored in program and considered invalid; will do nothing
- Due to counter based approach to debouncing used, all buttons must be released before new input is allowed

Final Note:

- Specification error: Due to misunderstanding forum discussion, parallel input was implemented different to expectations (Only same direction parallel input from local control implemented for different windows)

Fix: To achieve parallel input including requests in opposite directions from different windows in local control:

1. Use 3 request registers (Following format of 3 types of requests above) rather than the default 1 request register
2. Expand polling / debouncing method to debounce entire keypad, using same method of clearing bits that are pressed in the request register but now applying to one of 3 request registers depending on the column
3. At end of polling, if input was detected, push 2 requests rather than 1. There will be 2 cases:
    - 1. Central Control requested
        - In this case, central control buttons pressed so local control requests can be ignored. Push central control request and a clear register to reqQueue
    - 2. Local Control
        - In this case, central control buttons not pressed. In this case, push local increase request on reqQueue then local decrease request on reqQueue
4. When requests are to be executed, loop centralTimer twice to pop 2 requests off queue rather than the original 1 and execute both. If central control request is popped off queue, ignore the second request popped off reqQueue since empty (filler to ensure consistent incrementing of queueHead and queueTail).