Lab 2 - Virtual Timers

Goals

- Use the timer peripheral to get an interrupt
- Build a virtualized driver allowing any number of timers

Equipment

- Computer with build environment
- Micro:bit and USB cable

Documentation

- nRF52833 datasheet: https://docs-be.nordicsemi.com/bundle/ps_nrf52833/attach/nRF52833_PS_v1.7.pdf
    - Online version: https://docs.nordicsemi.com/bundle/ps_nrf52833/page/keyfeatures_html5.html
- Microbit schematic: https://github.com/microbit-foundation/microbit-v2-hardware/blob/main/V2/MicroBit_V2.0.0_S_schematic.PDF
- Lecture slides are posted to the Canvas homepage

Github classroom link: https://classroom.github.com/a/DUoodOEe

Lab 2 Checkoffs

You must be checked off by course staff to receive credit for this lab. This can be the instructor, TA, or PM during a Friday lab session or during office hours.

- **Part 2: Virtual Timers**
    a. Demonstrate your application reading the timer and printing it out
    b. Demonstrate your timer interrupt handler firing
    c. Demonstrate your timer callback turning on an LED
    d. Demonstrate your LED blinking due to repeated timer callbacks
    e. Demonstrate that your LED only blinks for 5 seconds
    f. Show your virtual timer code in virtual_timer.c
        - Including how you handle edge cases
    g. Demonstrate your final application with multiple repeating virtual timers

Also, don't forget to answer the lab questions assignment on Gradescope.

Lab Steps

Part 1: Setup

1. Find a partner

- Rule: you can pick any partner you want, but you can't pick the same partner twice
- You MUST work with a partner
    - If you can't find someone, as course staff for help

2. Create your Github assignment repo

- There is a github classroom link on the first page of this document. Click it!
- Pick a team name
- Pick your partner
- Generally, do what it says
- At the end, it should create a new private repo that you have access to for your code
    - Be sure to commit your code to this repo often during class!
- That link might 404. If it does, you first have to go to https://github.com/nu-ce346-student and join the organization
- **Important: both of you should join the repo before you can do the next step**

3. Set up an additional Git remote

- Open a terminal if you haven't yet
- `cd` into your "nu-microbit-base" repo
- At the top right of your shiny new private repo on the Github website, there is a green button that says "Code". If you set up an SSH key, you can click the SSH tab to get that URL, otherwise you should get the HTTPS URL. Either way, copy the URL so you can enter it into terminal
- `git remote add lab2 <YOUR-REPO-URL-HERE>`
    - This adds a "remote" repo hosted on github as a source for this repo
- **ONLY ONE OF YOU** should do the following steps
    - `git fetch lab2`
        - This gets the most recent commits from the new remote source
    - `git checkout lab2/main`
        - This changes your current commit to the remote source's main branch
    - `git switch -c lab2-code`
        - This makes a new branch for your lab code
    - `git push -u lab2 lab2-code`
        - This tells the new branch to push code to the new remote source

        - From now on, you can just pull, commit, and push as normal
- **THE OTHER STUDENT** should do this **AFTER** the first student finished the above steps:
    - `git fetch lab2`
    - `git switch lab2-code`
- **BOTH STUDENTS** should do this
    - `git submodule update --init --recursive`
        - Makes sure all git submodules are initialized and updated

4. Find the app starter files for this lab

- `cd software/apps/virtual_timers/`
    - This lab will use the files in this directory. Most of your changes will be in `main.c` and `virtual_timer.c` (with a few changes in `virtual_timer_linked_list.h`).

Part 2: Virtual Timers

1. Enable and Read a Hardware Timer

- Configure the TIMER4 peripheral.

  In `virtual_timer.c` inside the `virtual_timer_init()` function, you should configure the TIMER4 peripheral to be a **32-bit** timer that increments at **1 MHz**. Then you should clear the timer and start the timer.

  - Find the Timer peripheral in the nRF52833 manual and look at the registers.
  - You can access the TIMER4 peripheral registers with the code `NRF_TIMER4->REGISTER` (where REGISTER is replaced with the register name)
  - The Timer can be cleared with `NRF_TIMER4->TASKS_CLEAR = 1;`
    - And started with the `TASKS_START` register
  - For now you do not need to enable interrupts

- Implement the `read_timer()` function

  When a value of 1 is written to one of the `TASKS_CAPTURE[n]` register, the value of the timer is copied to the corresponding `CC[n]` register. Use the `TASKS_CAPTURE[1]` and `CC[1]` registers to implement the `read_timer()` function.

- Print the elapsed time from the timer

  Inside `main.c` inside the `main()` function, initialize TIMER4 and then print its value once per second. You can do this inside the while loop of `main()` with `nrf_delay_ms(1000)` and `read_timer()`.

- **CHECKOFF:** demonstrate your application printing out the elapsed time
  - *Question*: what are the units of the number that is printing?
  - Question: are the printed numbers what you expected?

2. Create a Non-Virtualized Timer

- You'll implement the steps for starting a timer in `virtual_timer.c` in the private `timer_start()` helper function

- Enable the `TIMER4` interrupt

  Choose a capture/compare channel that is *different* from the one you have already used to read the timer. You'll use this register as a Compare to trigger an interrupt.

  - In `virtual_timer_init()`, enable interrupts for this channel in the INTENSET register
    - Warning: read that register definition in the datasheet to figure out what bit to set!
  - Using `NVIC_EnableIRQ()`, enable interrupts for `TIMER4_IRQn` in the NVIC

- Use a single hardware timer

  Implement part of the `timer_start()` function such that an interrupt fires at the correct number of microseconds after this function call

  - An interrupt will fire when the value of the timer is equal to the value of `CC[n]` where n is the number of the capture/compare channel you enabled.
    - Be sure to set the `CC[n]` register to the duration *plus* the current timer value
    - If only there was a function you'd just written that got you the current timer value…
  - For now you can ignore the callback and `repeated` arguments of this function.
  - You can ignore overflow in this lab.

- Set a timer for two seconds in the future and print "Timer Fired!" in the interrupt handler when it occurs.
  - Note: You cannot call `timer_start()` directly as it is an internal helper function. The `virtual_timer_*` functions are external and *can* be used in `main.c`
  - You'll need to set the `cb` argument to *something*, but for now it can just be `NULL`.

- You might want to keep the once-a-second print of `read_timer()` from the last section, as it will be helpful when debugging.

- **CHECKOFF**: demonstrate your timer interrupt handler firing

3. Start Virtualizing a Single Timer

For now, we're just going to go through the virtualization process with a **single** timer. We'll handle multiple timers later after a few more improvements are made.

- Understand and modify the existing linked list node

  The linked list node is defined in `virtual_timer_linked_list.h`. Every node has a pointer to the next node as well as a timer value.

- Add a callback function field to the linked list node.

  Rather than placing code directly in the interrupt handler, we would like to call a callback function associated with each timer. A callback function is a function that is called when the event occurs. The application provides the desired callback function to your library as a function pointer it passes into `virtual_timer_start()`.

  So we can call the appropriate callback function when the event fires, we'll need to store it in the linked list node for the timer. The appropriate type is defined in `virtual_timer.h`: `virtual_timer_callback_t`.

  - Edit the linked list structure in `virtual_timer_linked_list.h` to add a field for the stored callback function pointer.

- Store the timer in a list

  Now edit the code in `timer_start()` to store information in the linked list.

  - Create a new linked list `node_t` using `malloc()`. The syntax is as follows:
    `my_type_t* my_type_pointer = (my_type_t*)malloc(sizeof(my_type_t));`

  - Store the timer expiration time in the linked list node as `timer_value`
    - This should be the duration *plus* the current timer value
  - Also store the callback function in the linked list node
  - Place the timer in the list using `list_insert_sorted()`
    - You may need to take a look at the helper functions in `virtual_timer_linked_list.h`

- Return a unique ID from `timer_start()`

  Each timer needs a unique identifier. Hint: `malloc()` returns a unique address each time you call it. You could use that address as an identifier.

- Call the associated callback when the timer interrupt fires

  When the timer fires, you should call the callback associated with that timer within the interrupt handler.

  - First you'll need to get the timer from the linked list using `list_remove_first()`
  - You can call the callback as: `my_timer_node->my_callback_variable();`
  - Don't forget to `free()` the node before returning
  - You might want to create a helper function to put all of this code into and call it from the interrupt handler.

- Add code to `main.c` to create a timer which toggles an LED on the Microbit

  - For now, you should only have one call to `virtual_timer_start()` in `main.c`
  - You should be using one of the callback functions, such as `led1_toggle()` to toggle the LED.

- **CHECKOFF:** demonstrate your timer callback turning on an LED
  - *Question:* what did you change in your code to make this work?

4. Create a Repeated Timer

- Modify `virtual_timer_linked_list.h` to add necessary information into the linked list node definition.
    - There are various things that you might add here.

- Modify `timer_start()` in `virtual_timer.c` to properly initialize the node based on whether the timer is repeated or not.

- Handle a repeated timer in the interrupt handler.

  You must detect if a timer is repeated, and if so update its expiration time in both the linked list and the compare register. Make sure to reinsert the timer into the linked list.

  Important: make sure you're re-inserting the same linked list node (with updated parameters) rather than `malloc`-ing a new node. That way the timer ID persists. So don't just call `timer_start()` in the interrupt handler.

  Although there are many ways to implement repeated timers, do NOT do so by resetting the hardware timer. While that solution would work for this particular problem, it does not scale to solving multiple virtual timers for later in the lab.

- In `main.c` create a repeated timer that toggles an LED on the Microbit every second.

- **CHECKOFF**: demonstrate your LED blinking due to repeated timer callbacks

5. Cancel the Repeated Timer

- Implement virtual_timer_cancel() in virtual_timer.c

  To cancel a timer, you must remove it from the linked list. Use the list_remove() function to accomplish this. Remember to free() the memory!

  You should also ensure that the capture/compare register is updated so that the interrupt does not fire for a removed timer.

- In `main.c` cancel the repeated timer that toggles the LED after five seconds have elapsed.

- **CHECKOFF**: demonstrate that your LED now only blinks for five seconds

6. Handle Multiple Virtual Timers

- Improve your implementation so it can handle multiple virtual timers.

  Currently, your implementation may not do so, but it is probably close. To handle multiple timers you must ensure that the compare register is always set to the soonest expiring timer in your linked list. Note that each of the timers may be repeating or not. You can continue ignoring overflow.

  You may want to check the functions available in `virtual_timer_linked_list.h` as some of them will be helpful for this larger implementation. Also be sure that you properly handle a list with no timers.

  - Properly set the capture/compare register to the soonest expiring timer in `timer_start()`
  - Properly set the capture/compare register to the soonest expiring timer when handling a timer interrupt
  - Properly set the capture/compare register to the soonest expiring timer when removing a timer from the list
  - Creating helper functions for repeated functionality is a good idea

  Note: You are only allowed to use a single `CC[n]` register for interrupts (and an additional register for reading the current time). The point is to virtualize a single `CC[n]` register so it can handle any number of timers.

  If you run into trouble, take a look at the `list_print()` function in virtual_timer_linked_list. Printing out the linked list contents once per second from the while loop in `main()` can be very informative.

- **Edge case:** timers firing very close to the same time

  Timers firing at very close to the same time and very short timers may be missed. Specifically, think about what may happen if you set the capture/compare register to the expiration time at the head of the list even when the timer's internal counter has already moved past this time.

  To catch this case, you need to make sure that you never exit the timer library, or set the capture/compare register, without handling and calling already-expired timers. Moving some code into a helper function may make this easier.

- **Edge case:** avoid concurrency issues

The linked list library is not safe for reentrancy, and your code probably isn't either. Consider the case where a timer occurs while you are inserting another timer. The checks for which value should be set in the capture/compare register could be incorrect (maybe you were dealing with the second timer in the list before the timer fired, but now it is the first timer in the list).

To prevent this, disable interrupts while modifying the linked list. You can use `__disable_irq()` and `__enable_irq()` to do so (that's two underscores before the function name). Interrupts that triggered while disabled will run when re-enabled. But now this means that the interrupt handler could be triggered even when the soonest expiring timer in the list should not be fired (because it was canceled). So you will need to check whether the head of the linked list should be expiring or not inside of the interrupt handler.

Make sure that you disable interrupts before *any* code that modifies the linked list and re-enable them afterwards. There may be multiple functions that you need to fix up. One function you don't need to fix is the interrupt handler itself. You won't be interrupted while the interrupt handler is running because interrupts don't preempt themselves (same priority!).

- **Edge case:** check for NULL pointers

  One additional concern is that the list could eventually become empty. Whenever you are interacting with the linked list, you should check whether the returned node is NULL. If you don't check for this properly, you'll get a hardfault when removing the last timer from the list...

- Add code to main.c which starts three separate repeating timers, each of which should control a single LED. Set one timer for 1 second, one for 2 seconds, and one for 4 seconds in duration. After 16 seconds, cancel the 2-second and 4-second timers. After four more seconds, cancel the 1-second timer.

- **Checkoff**: explain your code in virtual_timer.c to course staff
  - Including how you handle edge cases

- **Checkoff**: demonstrate your main.c application to course staff

Heads up: this lab can't be used in your final projects. And you wouldn't want to anyways. There's a decent library that virtualizes timers in the SDK. You should use that instead (APP_TIMER), and we'll use it in future labs.