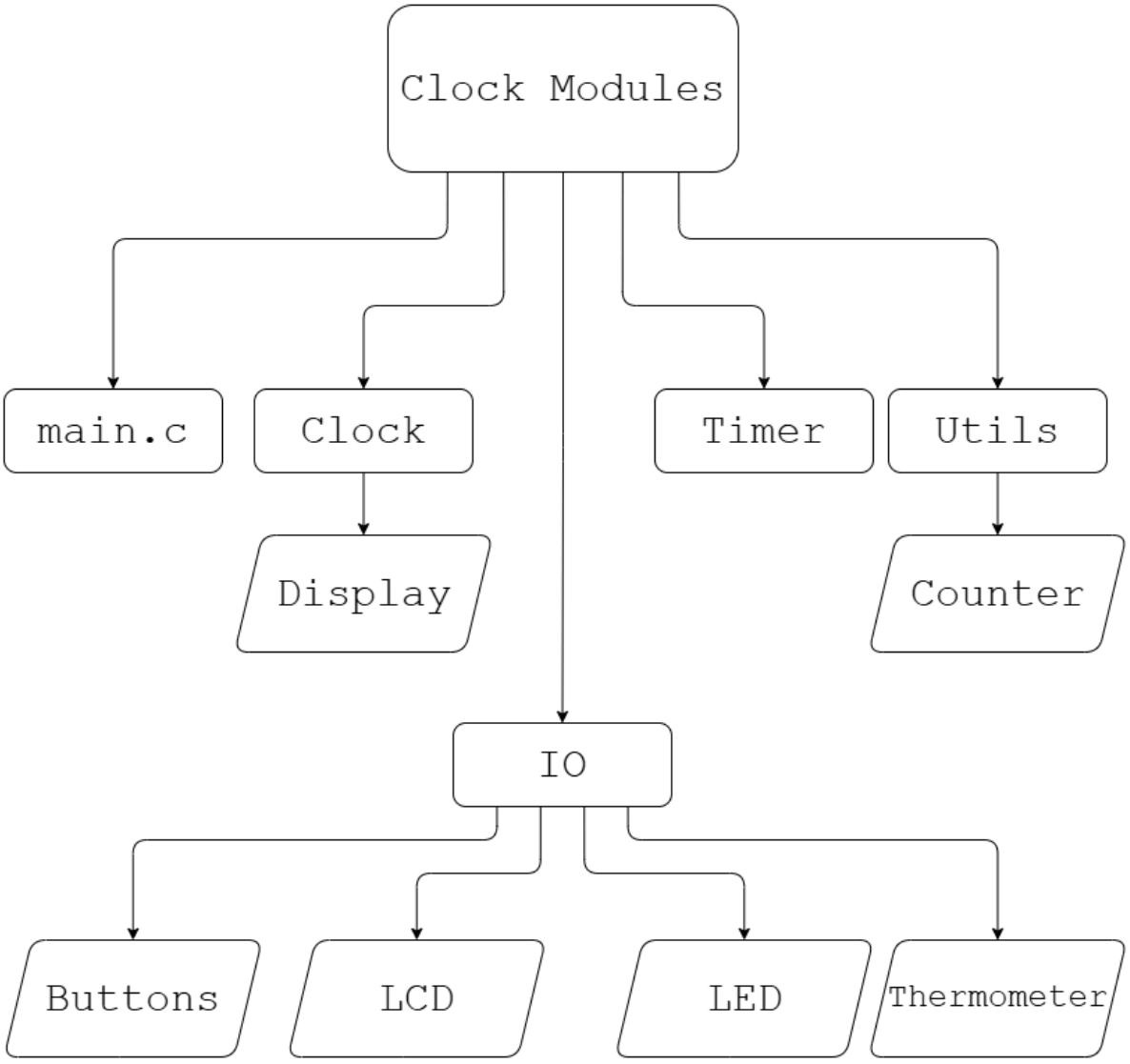
CA Lab2 Report

Q, Queue
Mackerels!

Contents

I Modules Overview

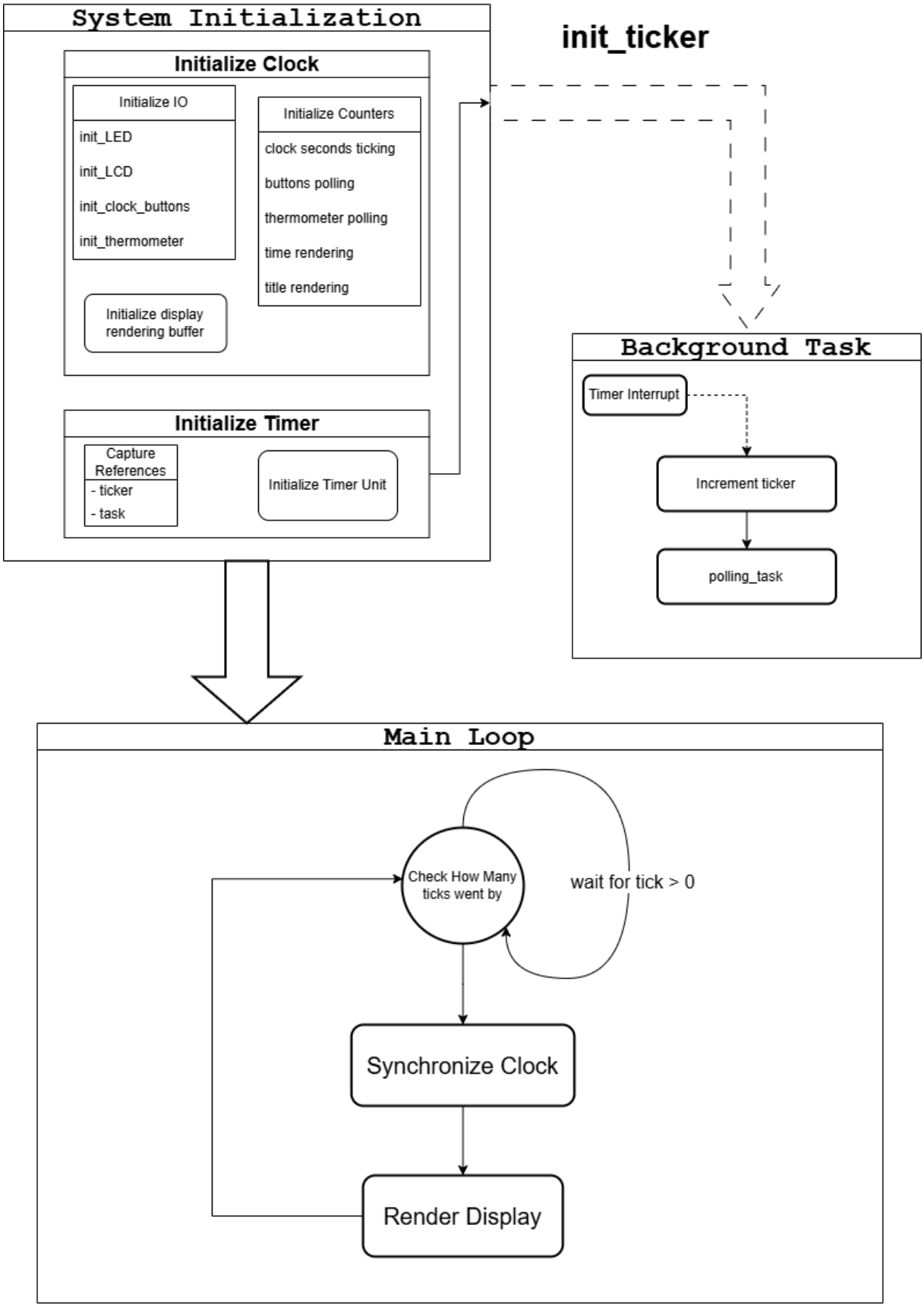


II Program Main Function

- This clock implementation model can accommodate and recover from tasks up to `2550ms` with losing track of time.
  - As long as the long running task allow for some time afterwards
  - In that case the rendering of the title will be effected! And time between switching will be longer than `10s`

```
#include <hidef.h>                          // Common defines

#include "clock.h"
#include "timer.h"

void main(void)
{
    volatile unsigned char timer_ticks = 0;
    unsigned char clock_event = 1;
    EnableInterrupts;                       // Global interrupt enable

    init_clock();

    init_ticker(&timer_ticks,polling_task); // towards semaphore-ish behavior.

    for(;;)                                 // Endless loop
    {
        // this loop doesn't have a fixed run time!
        // need to use semaphore to synchronies with the system clock
        // even that nothing is running in parallel!

        while (timer_ticks > 0)             // catch-up loop
        {
            timer_ticks--;                  // synchronize with timer clock
            ticking_task();
            clock_event = 1;                // one or more clock went by
        }

        if (clock_event)                    // allowed to skip a beat
        {
            // need to give ticking_task time to recover
            clock_event = 0;                 // event handled
            rendering_task();
            // if multiple ticks went by only to render screen/poll thermometer once
        }
    }
}
```

II. 1 Main Loop



III Clock Implementation

III. 1 Clock Configuration Header `clock.h`

Contains clock configuration.

III. 1.1 Clock Starting Time:

- `CLOCK_INITIALIZED_HOURS`
- `CLOCK_INITIALIZED_MINUTES`
- `CLOCK_INITIALIZED_SECONDS`

III. 1.2 Clock AM-PM mode initial state which is enabled by default

- `ENABLED_AM_PM_MODE`
- This functionality can be changed in runtime
- In SIMULATOR the `PTH7` is used as a button binding to toggle this state `on/off`

III. 1.3 Ticking/Polling/Rending rates as multiple of `SYSTEM_CLOCK_INTERVALS` which is `10ms`

- `CLOCK_TICKING_RATE`
- `BUTTONS_POLLING_RATE`
- `THERMOMETER_POLLING_RATE`
- `LCD_TIME_RENDING_RATE`
- `LCD_TITLE_RENDING_RATE`

III. 1.4 To achieve longer waiting time than what a 8-bit value can account for the `LCD_TITLE_RENDING_RATE` builds on the `LCD_TIME_RENDING_RATE`

- In this case on the `50th` time rendering trigger the title will change
- time is updated once every `200ms` which is `10ms * 20 = 200ms`
- title is changed every `10 S` which is `10ms * 20 * 50 = 10000ms`

```
// Clock Initial Starting Time
#define CLOCK_INITIALIZED_HOURS     11
#define CLOCK_INITIALIZED_MINUTES   59
#define CLOCK_INITIALIZED_SECONDS   45
// ----------------------------
/******** Clock Config ********/
// ----------------------------
// values of counter is multiple of SYSTEM_CLOCK_INTERVALS defined in timer.h as 10ms
#define CLOCK_TICKING_RATE       100
#define BUTTONS_POLLING_RATE     30
#define THERMOMETER_POLLING_RATE 20
#define LCD_TIME_RENDING_RATE    20
#define LCD_TITLE_RENDING_RATE   50
    // Title refresh rate is a multiple of Time one!
#define ENABLED_AM_PM_MODE 1
```

III. 1.5 Tasks Declaration

- `polling_task` buttons polling task to be done with the interrupt.
- `ticking_task` runs once per second and adjusted depending on clock mode.
- `rendering_task` update LCD display and also poll temperature with a countdown

```
void polling_task(void);
void ticking_task(void);
void rendering_task(void);
// ----------------------------
void init_clock(void);
```

III. 2 Clock Code Implementation

III. 2.1 Clock Global Variables

All global variables defined in `clock.c` are only accessible from within the `clock.c` translation unit and are not exported to the other modules.

- The clock needs to keep track current time for which three global variable `hours`, `minutes`, `seconds` and `temperature` is updated when polling the thermometer.
- The time is saved in `24-Hours` format regardless from the AM-PM Display Mode. The value of the hours will be adjusted accordingly when in the `time-render.c` module.
- Software Counters
  - `clock` ticks once per second if in `clock-ticking-mode`
  - `buttons_polling` triggers the buttons polling callback specified in `clock-buttons`. The interrupt manages this countdown to keep regular polling time intervals.
  - `thermometer_polling` update the temperature `temperature-polling`
  - `time_rending` update the time and temperature on the second LCD line
    - `title_rending` update the title once every 10 Seconds

```
// values assumes 10ms tact.

static unsigned char hours   = CLOCK_INITIALIZED_HOURS;
static unsigned char minutes = CLOCK_INITIALIZED_MINUTES;
static unsigned char seconds = CLOCK_INITIALIZED_SECONDS;

static int temperature;    // doesn't make much sense to initialize

// ------------

// Time Keeper / External Clock / should be in 10ms tact!

static Counter clock;

static Counter buttons_polling;

static Counter thermometer_polling;

static Counter time_rending, title_rending;
```

III. 2.2 Clock Operating Modes

III. 2.2 .a Ticking Mode

In this mode the seconds are incremented on the 100th tick countdown with `10ms` intervals.

- When overflow occurs will trickle down to the increment the `minutes` and eventually the `hours` variable.

```
#define TICKING_MODE_LED_MASK 0x01
static void ticking_mode(void)
{
    toggle_LED(TICKING_MODE_LED_MASK);
    seconds++;
    if (seconds >= 60)
    {
        seconds = 0;
        minutes++;
        if (minutes >= 60)
        {
            minutes = 0;
            hours++;
            if (hours >= 24)  hours = 0;
        }
    }
}
```

III. 2.2 .b Set Mode

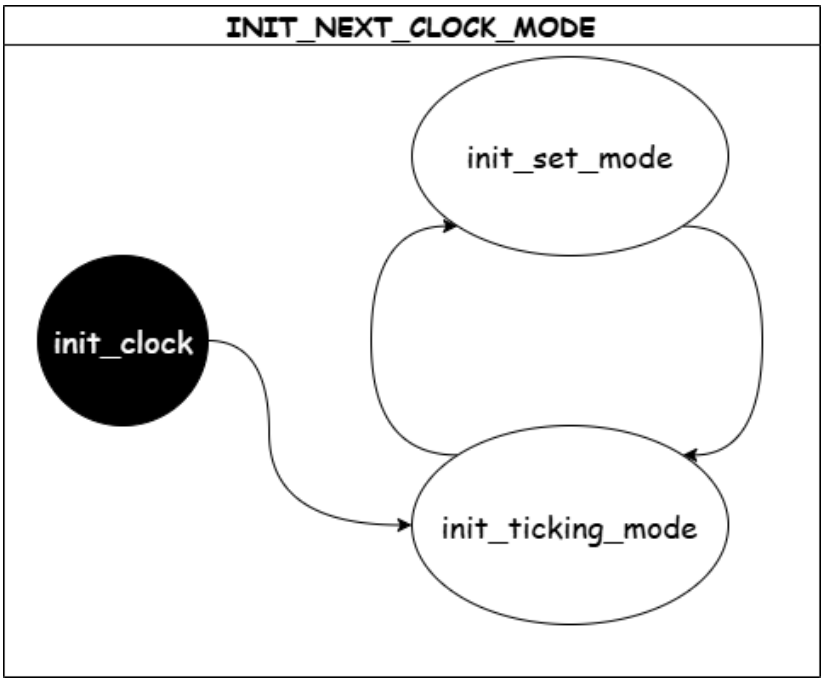
In the set mode the clock will not be ticking. The buttons polling is done from within the interrupt to guarantee consistent polling time.

III. 2.2 .c Switching Clock Modes

An initialization callback is triggered each time a clock switch modes

- A Mutating Function Pointer is used!
  - `INIT_NEXT_CLOCK_MODE` is a function pointer that changes after each call!
  - Each clock mode also defines a init function where the next `INIT_NEXT_CLOCK_MODE` is reassigned.
- The enabled buttons is toggled to mask/unmask the increment buttons.
  - This assumes that the initial clock mode is the ticking mode
    - If the clock is to be initialized in set mode, need to change the enabled buttons in `buttons-initialization!`

```
// defines a pointer to a void function(void) as a type
typedef void (*Callback)(void);

// defines a pointer to a function that returns a function pointer
typedef Callback (*CallbackInitializer)(void);
// self mutating/mutilating references
// // changes actual subroutine after each call!
static volatile CallbackInitializer INIT_NEXT_CLOCK_MODE;

static void switch_clock_mode(void)
{
    clock.callback = INIT_NEXT_CLOCK_MODE(); // for the next clock event!
}
```



III. 2.2 .c .a Initialize Ticking Mode

- Sets the next initialization function.
- Sets the LED state
- Rewind the clock ticker to start of the second
- Returns the corresponding mode callback to update the counter callback `clock.callback`

```
static Callback init_set_mode(void); // declaration needed for cyclical reference

#define TICKING_MODE_LED_MASK 0x00

static Callback init_ticking_mode(void)
{
    INIT_NEXT_CLOCK_MODE = init_set_mode;
    disable_buttons(ENABLE_PTH0 | ENABLE_PTH1 | ENABLE_PTH2);
    set_LED(TICKING_MODE_LED_MASK);  // turn off LED
    rewind(&clock); // reset clock ticker to start at the start of the second
    return ticking_mode;
}
```

III. 2.2 .c .b Initialize Set Mode

- Sets the next initialization function.
- Sets the LED state
- Returns the corresponding mode callback to update the counter callback `clock.callback`

```
#define SET_MODE_LED_MASK 0x80

static Callback init_set_mode(void)
{
    INIT_NEXT_CLOCK_MODE = init_ticking_mode;
    enable_buttons(ENABLE_PTH0 | ENABLE_PTH1 | ENABLE_PTH2);
    set_LED(SET_MODE_LED_MASK);
    return set_mode;
}
```

III. 2.3 Clock Initialization

The clock need to:

- Initialize the Input/Output Peripherals
  - `init_LED`
  - `init_LCD`
  - `init_thermometer`
  - `init_clock_buttons` initialize the buttons register and bind corresponding callbacks
- Initialize the Display render
  - Initialize the second LCD line internal buffer for time/temperature representation
- Initialize the counter with the callback binding and the reset values as configured in the `clock.h`

```
static Counter clock;

static Counter buttons_polling;

static Counter thermometer_polling;

static Counter time_rending, title_rending;

void init_clock(void)
{
    // -------------
    /** Init IO ***/
    // -------------

    init_LED();
    init_LCD();
    init_clock_buttons();
    init_thermometer();

    // ------------

    init_render();

    // ------------

    init_counter(&clock, CLOCK_TICKING_RATE, init_ticking_mode());

    init_counter(&buttons_polling, BUTTONS_POLLING_RATE, poll_buttons);

    init_counter(&thermometer_polling, THERMOMETER_POLLING_RATE, poll_temperature);

    init_counter(&time_rending, LCD_TIME_RENDING_RATE, lcd_rendering_callback);

    // triggered on the nth time the time_reding resets!
    init_counter(&title_rending, LCD_TITLE_RENDING_RATE, render_title);
}
```

III. 2.3 .a Clock Buttons Initialization

- `init_buttons` must be called first before binding callback in the `BUTTONS_CALLBACK_REGISTRAR`
  - By default `PTH3` is used to switch clock mode should never be masked
  - The `PTH7` toggles the AM-PM state by calling `toggle_am_pm` which is defined in `toggle-am-pm`
    - There is no corresponding physical buttons on the board this is only useable in the SIMULATOR

```
static void init_clock_buttons(void)
{
    init_buttons(ENABLE_PTH3 | ENABLE_PTH7);

    BUTTONS_CALLBACK_REGISTRAR[PTH3_TABLE_ENTRY] = switch_clock_mode;
    BUTTONS_CALLBACK_REGISTRAR[PTH7_TABLE_ENTRY] = toggle_am_pm;

    BUTTONS_CALLBACK_REGISTRAR[PTH2_TABLE_ENTRY] = inc_hours;
    BUTTONS_CALLBACK_REGISTRAR[PTH1_TABLE_ENTRY] = inc_minutes;
    BUTTONS_CALLBACK_REGISTRAR[PTH0_TABLE_ENTRY] = inc_seconds;
}
```

III. 2.3 .b Increment Buttons Definitions

- The increment doesn't carries over.
- A reference to the callbacks is registered in the `BUTTONS_CALLBACK_REGISTRAR`
- Masked in ticking mode and will be ignored except in set mode

```
static void inc_hours(void)
{
    hours++;
    if (hours >= 24)
        hours = 0;
}

// ------------

static void inc_minutes(void)
{
    minutes++;
    if (minutes >= 60)
        minutes = 0;
}

// ------------

static void inc_seconds(void)
{
    seconds++;
    if (seconds >= 60)
        seconds = 0;
}
```

III. 2.4 Counter Callbacks

III. 2.4 .a Temperature Polling

- Update the global variable `temperature`
- Wrapped in a callback to be be passed to a software counter

```
static void poll_temperature(void)
{
    temperature = poll_thermometer();
}
```

III. 2.4 .b Display rendering Countdown

The clock interface with `time-render.c` by calling `render_time` and passes required values as parameter to avoid exporting global variables.

- The title rending counter piggy-back on the time rendering so that on the n_{th} the time is updated the title changes too.
  - set to be 50 times the time rending rate which is once per `200ms`

```
static void lcd_rendering_callback(void)
{
    render_time(
        hours,
        minutes,
        seconds,
        temperature
    );
    // dependent countdown
    countdown(&title_rending);  // and tick down title counter
}
```

- In the clock init function the callback and the reset value are set.

```
init_counter(&time_rending, LCD_TIME_RENDING_RATE, lcd_rendering_callback);

// triggered on the nth time the time_reding resets!
init_counter(&title_rending, LCD_TITLE_RENDING_RATE, render_title);
```

III. 3 Clock Tasks

Tasks are called from the `main.c` modules which tries to synchronizes with the timer interrupt.

III. 3.1 Polling Task

Noticeable by the end user if not done with exact timing.

Need to be done from inside the interrupt.

```
void polling_task(void)
{
    // must be done in regular intervals -> called from within the interrupt
    countdown(&buttons_polling);
}
```

III. 3.2 Ticking Task

Must catch up to the interrupt clock in ticking mode.

```
void ticking_task(void)
{
    // allowed to lag a bit behind. But need to catch up eventually!
    countdown(&clock);
}
```

III. 3.3 Rending Task

Can deal with more latency and even skip a couple of ticks

```
void rendering_task(void)
{
    countdown(&thermometer_polling);
    countdown(&time_rending);
}
```

IV Clock Display

Defines how the title/time/temperature to be displayed on the LCD Screen.

Interface with the LCD driver functions defined in `lcd.h`

IV. 1 Time Render

- Needs to have `ENABLED_AM_PM_MODE` defined to set AM-PM mode.
- Must be initialized first by calling `init_render` to setup the internal buffer.
- Defines the `render_time_function` which is called from the `clock.c` and given the time and temperature as parameters.

IV. 1.1 Time Render Global Variables

- `AM_PM_MODE` AM-PM State
- `TIME_LINE` Buffer to be flushed on the second LCD line when `render_time` is called
  - Buffer size is defined by the `LCD_LINE_WIDTH` defined in `lcd.h`
  - The internal buffer offsets are labeled to allow the time/temperature to be updated independently form one another.
  - The buffer is split into two section to allow for different `LCD_LINE_WIDTH` values.
    - right aligned clock representation
    - left aligned temperature representation
      - A minium line width of 15 must be accounted for otherwise temperature and time will overlap!

```
static unsigned char AM_PM_MODE = ENABLED_AM_PM_MODE;

static char TIME_LINE[LCD_LINE_WIDTH];

// right aligned
    #define hours_str             (TIME_LINE)
    #define hm_separator          (TIME_LINE + 2)
    #define minutes_str           (TIME_LINE + 3)
    #define ms_separator          (TIME_LINE + 5)
    #define seconds_str           (TIME_LINE + 6)
    #define AM_PM_str             (TIME_LINE + 8)

// left aligned
    #define temperature_str       (TIME_LINE + LCD_LINE_WIDTH - 5)
    #define temperature_unit_str  (TIME_LINE + LCD_LINE_WIDTH - 2)
```

IV. 1.2 Initialize Time Rendering Buffer

- `TEMPERATURE_GRADE_ENCODING` is specified in `lcd.h` with different values for different compilation targets

```
void init_render()
{
    repeat_char(TIME_LINE, ' ', LCD_LINE_WIDTH);   // initialize buffer line

    hm_separator[0] = ':';
    ms_separator[0] = ':';

    temperature_unit_str[0] = TEMPERATURE_GRADE_ENCODING; // defined in lcd.h

    temperature_unit_str[1] = 'C';
}
```

IV. 1.3 Updating Rendered Time

Uses `decToASCII`, `signed/unsigned` functions defined in `ASCII-Utils.c` to convert numbers into decimal ASCII representation.

Expect values that is to displayed to be passed as parameters.

```
void render_time(
    unsigned char hours,
    unsigned char minutes,
    unsigned char seconds,
    int temperature
){
    unsigned_decToASCII(
        represent_hours(hours),
        hours_str,
        2
    );

    unsigned_decToASCII(
        minutes,
        minutes_str,
        2
    );

    unsigned_decToASCII(
        seconds,
        seconds_str,
        2
    );

    signed_decToASCII(
        temperature,
        temperature_str,
        2
    );

    write_line(TIME_LINE, 1);
}
```

IV. 1.4 Toggle AM-PM

- A global variable is used to keep track of current am-pm mode.

```
static unsigned char AM_PM_MODE = ENABLED_AM_PM_MODE;
```

- Toggling the AM-PM mode will always sets the `AM_PM_str` as empty spaces. This will be accounted for `represent_hours` function

```
void toggle_am_pm(void)
{
    AM_PM_str[0] = ' ';
    AM_PM_str[1] = ' ';

    AM_PM_MODE = !AM_PM_MODE;
}
```

IV. 1.4 .a Represent Hours

Hours representation need to be handled by a function to allow toggle am-pm mode in runtime

```
static unsigned char represent_hours(unsigned char hours)
{
    if (!AM_PM_MODE) return hours;

    // side effect!
    AM_PM_str[0] = hours < 12 ? 'A' : 'P';
    AM_PM_str[1] = 'M';      // assumes morning

    if (hours < 13)
    {
        if (hours == 0) return 12U;
        return hours;
    }
    return hours - 12;
}
```

IV. 2 Title Render

- Writes a title to second LCD line by calling `render_title` function.
- Cycles through Titles in the array `TITLES` from the header file `title-render.c` each time the `render_title is called`.

IV. 2.1 Title Renderer Implementation

- One or more Title is defined in the `TITLES` Array.
  - Titles will be cycled each time the title is needed to be updated
- Three global pointer variable are needed to keep track of current/next title
  - `starting_title` points to the first string in the array. Also needed to rewind the titles.
  - `current_title`
  - `titles_boundary` points to end of the arrays

```
const char* TITLES[] = {
    "(C) IT SS2025",
    "Q, Queue",
    "Mackerels!"
};
// ----------------------------
#define SIZEOF(Array) (sizeof(Array) / sizeof(Array[0]))

static char** starting_title = TITLES;
static char** current_title = TITLES;
static char** titles_boundary = TITLES + SIZEOF(TITLES);

// ----------------------------

void render_title(void)
{
    // render current title and shift the pointer for next title
    write_line(*current_title++, 0);

    // check of out of bound to rewind if needed
    if (current_title >= titles_boundary)
        current_title = starting_title;
}
```

IV. 3 ASCII-Utils

IV. 3.1 decToASCII

IV. 3.1 .a unsigned_decToASCII

- Parameters:
  - `unsigned int number` to be converted
  - `char* at` starting posting on a string to start writing the digits
  - `unsigned char digits` number of digits to write
- Converts a 16-bit unsigned integer to decimal ASCII representation
- Writes n lower decimal digits into a string.
- Writes leading zeros if number requires less digits than specified.
- Does not adds null termination!

```
void unsigned_decToASCII(unsigned int number, char* at, unsigned char digits)
{   // writes leading zeros
    unsigned int rest;

    while (digits-- > 0)
    {
        rest = number % 10;
        at[digits] = rest + '0';
        number /= 10;
    }
}
```

IV. 3.1 .b signed_decToASCII

- Parameters:
  - `int number` to be converted
  - `char* at` starting posting on a string to start writing the digits
  - `unsigned char digits` number of digits to write
  - requires one more char for the sign!
- Converts a 16-bit integer to decimal ASCII representation
- Write either a negative sign or empty space for positive numbers.
- Writes n lower decimal digits into a string.
- Writes leading zeros if number requires less digits than specified.
- Does not adds null termination!

```
void signed_decToASCII(int number, char* at, unsigned char digits)
{ // requires one extra space for the sign
    at[0] = ' ';
    if (number < 0)
    {
        at[0] = '-';
        number = -number; // ~number +1
    }
    unsigned_decToASCII(number, at +1, digits);
}
```

IV. 3.1 .c repeat_char

- Parameters:
  - `char* str` starting position of the string
  - `char c` character to be repeated
  - `unsigned char length` length of string/repeat count
- used to initialize a string buffer with a specific value.
- Does not adds null termination!

```
void repeat_char(char* str, char c, unsigned char length)
{
    while (length--) *str++ = c; // countdown loop
}
```

V Clock Timer

Defines the timer interrupts behavior.

- The timer code was rewritten in C to provide better cohesion with the rest of the code base.

V. 1 Timer Implementation

V. 1.1 Timer External References

The timer module needs to capture to external references:

```
// value will be incremented on each timer trigger
static volatile unsigned char* ticker;
    // external reference to be captured at initialization

static void (*in_sync_callback) (void);
```

- `ticker` 8-bit variable which is incremented on each interrupt
- `in_sync_callback` a callback with hard time deadline with low latency tolerance which will be called on each interrupt

V. 1.2 Timer Channel Configuration

The timer is setup to the the 4th timer channel in output capture mode.

```
#define ENABLE_TIMER_UNT       0x80

// commenting in code form!
#define TIMER_CH4              0x10
#define TIMER_CH               TIMER_CH4
#define PRESCALE_FACTOR        0x07
#define TC                     TC4
#define TCTL_REGISTER          TCTL1

// needs two mask to sets the mode properly!
#define TCTL_MODE_AND_MASKING  0xFC   // set lower two bits to 0 and leaves the rest!
#define TCTL_MODE_OR_MASKING   0x00   // don't set any bits to 1!

// ----------------------------

static void init_timer_uint()
{
    TSCR1 = ENABLE_TIMER_UNT;

    TSCR2 |= PRESCALE_FACTOR;

    TC += SYSTEM_CLOCK_INTERVALS;    // setup next interrupt timer

    TIOS  |= TIMER_CH;    // set as ouput capture mode on the timer channel used

    // sets bits to zero where there are zeros in the mask
    TCTL_REGISTER = TCTL_REGISTER & TCTL_MODE_AND_MASKING;

    // sets bits to one where there are ones in the mask
    TCTL_REGISTER = TCTL_REGISTER | TCTL_MODE_AND_MASKING;

    TIE   |= TIMER_CH;    // enables interrupt on the timer channel
}
```

V. 1.3 Timer ISR Implementation

The timer is implemented as an interrupt service routine which is triggered by the hardware after `SYSTEM_CLOCK_INTERVALS` ticks elapsed as specified external cpu bus clock and the prescale divider factor.

```
interrupt 12 void TimerISR(void)
{
    TC += SYSTEM_CLOCK_INTERVALS;    // setup next interrupt timer
    TFLG1 |= TIMER_CH;       // clears the interrupt flag

    *ticker += 1;         // indicates how many NEXT_TIMER_TRIGGER have passed

    in_sync_callback();  // hard real time tasks
}
```

V. 2 System Clock

Defines how often the interrupt is triggered set be `10ms`.

All counters are configured as multiple of this base interrupt interval.

V. 3 Timer Initialization

Function: `init_ticker`

- Parameters:
  - `volatile unsigned char* ticker` capture reference to a ticker variable to be incremented on each interrupt
  - `void (*hard_real_time_task) (void)` low latency tolerance callback which will be triggered on each interrupt

```
void init_ticker(
    volatile unsigned char* referenced_ticker,
    void (*hard_real_time_task) (void)
){
    // capture reference
    ticker = referenced_ticker;
    in_sync_callback = hard_real_time_task;
    init_timer_uint();
}
```

V. 4 Timer Initialization Usage Example

```
volatile unsigned char ticks;

void poll_buttons(void);

init_ticker(&ticks, poll_buttons);
```

VI Software Counter

- Implements a software counter that:
  - Countdowns to zero from a starting value
  - When counter hits zeros it resets back to the starting value
  - On Reset a callback is triggered

VI. 1 Counter Object Like Implementation

Defined as structure `Counter` that is meant emulate object orient use-case:

- By holding a state `ticker` aka. `#attribute`
- And defining function `countdown` that modify the `object` state `#method`

This approach is more or less what the Go Programming Language takes to implements objects oriented programming features and what is inspiration for the `Counter` module.

- Methods in Go

VI. 2 Structured Counter

The `Counter` structure holds the following information:

- `ticker` current counter value.
- `reset` reset value when the `ticker` hits zero.
- `callback` a void function pointer to be called on reset.

```
typedef struct {                // not opaque!

    unsigned char ticker;       // ticking counter until next rest
    // counts how many external clock ticks must go by until reset/trigger

    unsigned char reset;        // periodic countown trigger value

    void (*callback) (void);    // triggered on reset

} Counter;
```

VI. 3 Counter Functionality

The `counter.h` module defines the following operation `"methods"` on the `Counter` structure `"object"`.

VI. 4 Initialize Counter

Function: `init_counter`

Initialize a counter structure to abstract some `Counter` implementation details.

- Parameters:
  - `Counter* counter` Pointer to the structure instance.
  - `unsigned char reset_on` Specify when the `counter` to trigger a reset
  - `void (*subroutine) (void)` A callback which runs on rest

VI. 4.1 Initialize Counter Implementation

- The `counter` is initialized to be triggered on the first `countdown` call
- The `reset_on` value refers to how many time a countdown need to happens until reset. `reset on the n-th tick`
  - `reset_on` value of 1 means reset on every countdown.

```
void init_counter(
    Counter* counter,
    unsigned char reset_on,
    void (*subroutine) (void)
){
    counter->ticker   = 0;                  // set to trigger at next countdown
call // used as initializer
    counter->reset    = reset_on -1;        // triggers on the n th tick. resets at
zero!
    counter->callback = subroutine;         // called on reset
}
```

VI. 4.2 Initialize Counter Usage Example

```
#define BUTTONS_POLLING_RATE     30

void poll_buttons(void);

Counter buttons_polling;

init_counter(&buttons_polling, BUTTONS_POLLING_RATE, poll_buttons);
```

VI. 5 Counter Countdown

Function: `countdown`

- `countdown` check the current value of the ticker `attribute`
  - After checking `counter->ticker` value will be decremented.
  - If the `ticker` value is zero a reset will be triggered.
  - In case of reset the `ticker` value is assigned the `counter->reset` to count down again from there.
  - A callback is called on reset

VI. 6 Counter Countdown Implementation

```
void countdown(Counter* counter)
{
    if (counter->ticker-- != 0) return;     // no reset yet

    counter->ticker = counter->reset;

    counter->callback();                    // triggers callback on reset
}
```

VI. 7 Counter Countdown Usage Example

```
countdown(&buttons_polling);
```

VI. 8 Rewind Counter

Just a counter reset without a callback triggered.

VII Buttons

Provide a way to interface with input buttons on the Dragon Board12.

VII. 1 Buttons Initialization

The `init_buttons` function must be called at the start before configuring the buttons with the initial enabled buttons state.

- To enable a button labels are defined as `#define` constant which corresponds to bit positional masks.

```
#define ENABLE_PTH0          1U
#define ENABLE_PTH1          2U
#define ENABLE_PTH2          4U
#define ENABLE_PTH3          8U
#define ENABLE_PTH4          16U
#define ENABLE_PTH5          32U
#define ENABLE_PTH6          64U
#define ENABLE_PTH7          128U
```

- To enable multiple buttons bitwise or operator `|` is needed with the enable buttons label.
- In the initialization phase:
  - the buttons callback is set to an empty function.
  - The The Data Direction Register of port `H` is set as an input register.

```
void init_buttons(unsigned char enable_initial_state)
{
    unsigned char i;
    for (i = 0; i < BUTTONS_COUNT; i++)
        BUTTONS_CALLBACK_REGISTRAR[i] = UNMAPPED;

    DDRH = 0x00;    // Configure Port H as input register
    enabled = enable_initial_state;
}
```

- Usage Example:

```
   init_buttons(ENABLE_PTH3 | ENABLE_PTH7);
```

VII. 2 Enabled Buttons Mask

A global variable `enabled` is used to mask buttons.

```
static volatile unsigned char enabled = 0x00;
```

- The variable `enabled` is only defined in the buttons.c translation unit to protect it and make it only accessible by intended functions.
- When polling the buttons the enabled mask is used in an and mask to ignore a button press on a disabled button
- Due to reverse polarity `active-low/active-high` when compiling for `SIMULATOR/MONITOR` targets an assembler directive is used to change the function like macro definition

```
#ifdef SIMULATOR
        #define poll_buttons_state() (PTH)    // inlined
    #else
        #define poll_buttons_state() (~PTH)
#endif
```

VII. 3 Buttons Call Back Registrar

- `init_buttons` must be called first!

Array of function pointers to bind a callback when a buttons is pressed.

- A label is used to specify the button table entry like `PTH0_TABLE_ENTRY`

```
static void init_clock_buttons(void)
{
    init_buttons(ENABLE_PTH3 | ENABLE_PTH7);

    BUTTONS_CALLBACK_REGISTRAR[PTH3_TABLE_ENTRY] = switch_clock_mode;
    BUTTONS_CALLBACK_REGISTRAR[PTH7_TABLE_ENTRY] = toggle_am_pm;

    BUTTONS_CALLBACK_REGISTRAR[PTH2_TABLE_ENTRY] = inc_hours;
    BUTTONS_CALLBACK_REGISTRAR[PTH1_TABLE_ENTRY] = inc_minutes;
    BUTTONS_CALLBACK_REGISTRAR[PTH0_TABLE_ENTRY] = inc_seconds;
}
```

VII. 4 Enable Buttons

- Enable buttons at bit position specified by a bit-select-mask

```
void enable_buttons(unsigned char mask)
{
    enabled |= mask;
}
```

VII. 5 Disable Buttons

- Disable buttons at bit position specified by a bit-select-mask

```
void disable_buttons(unsigned char mask)
{
    enabled &= (~mask);
}
```

VII. 6 Toggle Enabled Buttons

Use the XOR operator ^ with a bit select mask to toggle the enable state of the buttons like is required when switching clock modes.

```
void toggle_enabled_buttons(unsigned char mask)
{
    enabled ^= mask;
}
```

VII. 6.1 Usage Examples

```
    enable_buttons(ENABLE_PTH0 | ENABLE_PTH1 | ENABLE_PTH2);
    disable_buttons(ENABLE_PTH0 | ENABLE_PTH1 | ENABLE_PTH2);

    toggle_enabled_buttons(ENABLE_PTH0 | ENABLE_PTH1 | ENABLE_PTH2);
```

VII. 7 Buttons Polling

This is done by calling the `poll_buttons` function.

- First the current buttons state is polled.
  - To account for active `low/high` polarity state the register will be negated if needed depending on compilation target
  - The buttons register is masked be the enabled bit select mask to ignore disabled buttons
- A loop test the state of the buttons at a specific bit
  - If the bit is set to 1 this means the buttons is pressed
  - If a buttons is pressed a callback from the `BUTTONS_CALLBACK_REGISTRAR` is triggered
  - The `mask` variable shifted to test next bit position.

```
void poll_buttons(void)
{
    unsigned char i;

    unsigned char mask = 1;
        // shifted to test the buttons register at a specific bit

    unsigned char buttons = poll_buttons_state();
        // current buttons state normalized to be true if pressed independent from
compilation target
        // buttons state is masked by enabled buttons!

    for (i = 0; i < BUTTONS_COUNT; i++)
    {   // loops over all registered callbacks and call the ones with active button
state
        if (mask & buttons)
            BUTTONS_CALLBACK_REGISTRAR[i]();

        mask = mask << 1;
    }
}
```

VIII LCD Driver

Provides a way to interface with LCD Screen Device.

VIII. 1 LCD Initialization

Call `init_LCD` to setup the LCD.

VIII. 2 LCD Display Interface Functions

VIII. 2.1 Write Line

- Parameters:
  - `const char* text` the string to be written on LCD
  - `char line` the line on which the string will be written
- The use of `const` should have no effect on the functionality and will work when passed a non constant value. This is just a promise from the function not to change values passed to it as a pointer.

VIII. 3 Special Character Encoding

The display device has a special encoding for the temperature celsius grade encoding ° which is not in the original ASCII standard but form the extended one.

- The encoding differ depending on compilation target.

```
#ifdef SIMULATOR
    #define TEMPERATURE_GRADE_ENCODING 0xB0
#else
    #define TEMPERATURE_GRADE_ENCODING 0xDF
#endif
```

IX LED Interface

IX. 1 LED Initialization

The Function `init_LED` need to be called before using the LED to setup the Data Direction Registry on the `PORTB`

- The Seven Segments Display also needs to disabled when running on board otherwise will the lights might flicker.
  - Because this is not really a part of the clock. Disabling the Seven Segments Display can be done at the same time when initializing the LEDs

```
static void disable_seven_segment(void)
{
    // will flicker otherwise
    DDRP = 0x0F;    // Port P.3..0 as outputs (seven segment display control)
    PTP  = 0x0F;    // Turn off seven segment display
}

void init_LED(void)
{
    DDRJ_DDRJ1  = 1;    // Port J.1 as output
    PTIJ_PTIJ1  = 0;
    DDRB        = 0xFF;  // Port B as output
    PORTB       = 0x00;
    disable_seven_segment();    // just hide it here. doesn't really have place
elsewhere
}
```

IX. 2 set LED

Function: `set_LED`

Overwrites the LED register with a given value.

- Parameters:
  - `unsigned char value` value to overwrites the LED state on `PORTB`

```
set_LED(0x80);
```

IX. 3 get LED

Function: `get_LED`

returns the LED register state.

```
unsigned char LED_State = get_LED();
```

IX. 4 toggle LED

Function: `toggle_LED`

Uses a bit select mask to switch LED `on/off` at a specific bit position.

- Parameters:
  - `unsigned char mask` bit select to target a bit position to toggle LED with the xor operator.

```
toggle_LED(0x01);
```

X Thermometer Driver Interface

X. 1 Thermometer Initialization

Function: `init_thermometer`

Initialize the AC-DC required hardware circuities by writing to the corresponding registers.

```
void init_thermometer(void)
{
    ATD0CTL2 = 0xC0;        // Enable ATD
    ATD0CTL3 = 0x08;        // Single conversion only
    ATD0CTL4 = 0x05;
}
```

X. 2 Temperature Polling

Function: `poll_thermometer`

Returns temperature as signed 16-bit value.

X. 2.1 Temperature Polling Implementation

```
#define CONVERTING_CHANNEL      0x87
#define AC_DC_CONVERTING_BIT    0x80

int poll_thermometer(void)
{
    // start converting
    ATD0CTL5 = CONVERTING_CHANNEL;

    // wait until hardware flag resets when the AC-DC converting finishes
    while (ATD0STAT0 & AC_DC_CONVERTING_BIT != 0);

    // convert result as human readable unsigned temperature value
    return (ATD0DR0 * 50) / 511 - 30;
}
```

X. 2.2 Temperature Polling Usage Example

```
    int temperature = poll_thermometer();
```