Computer Architecture Lab 2

Elias und Basel

June 21, 2023

Abstract

Here we trying to document our Lab2 Homework of Computer Architecture lesson in Hochschule Esslingen.

1 Task 2.1

1.1 Fixing ticker.asm

In our defines we need to give TIMER_ON value $80 instead of $70 , aslo we need to give TSCR2 value #$07 so our instructions should look something like

```
TIMER_ON    equu  $80          ; tscr1 value to turn ECT on

MOVB #$07 ,  TSCR2             ;initiliaze TSCR2
```

1.2 Testing ticker.asm

Since we are working on simulator, we need to use CPU's Cycles to test our ticker.asm

So we can put a breakpoint in the isrECT4 method like shown in Figure 1

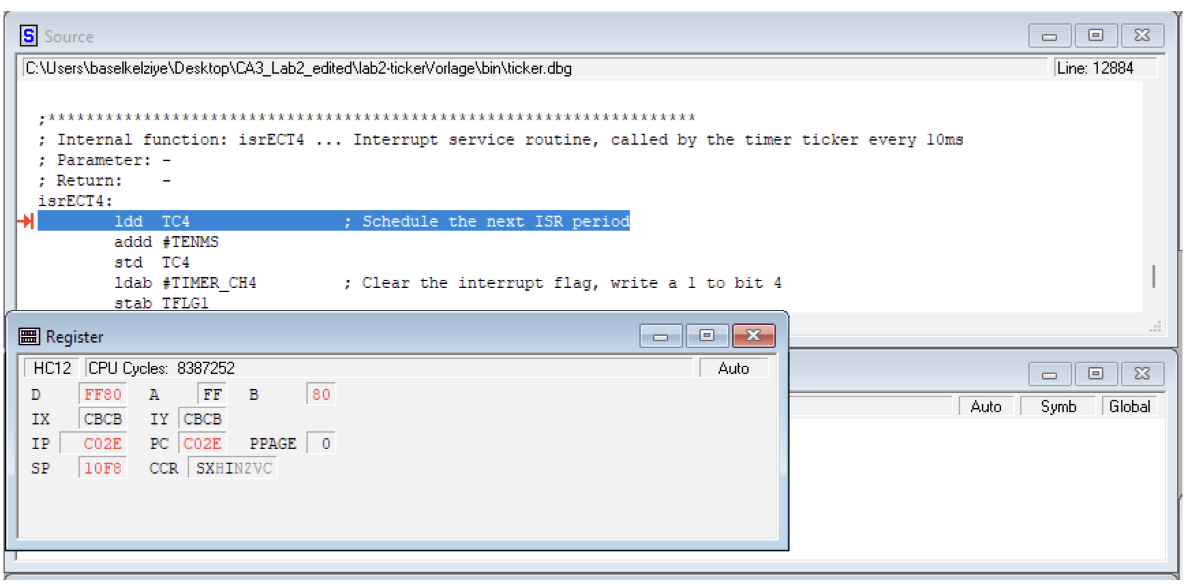


Figure 1: CPU Cycle After First Call

Here we can see that at the moment our cycle amount is 8387252.
Now lets start our program and let see the cpu cycle amount when we hit the breakpoint again.

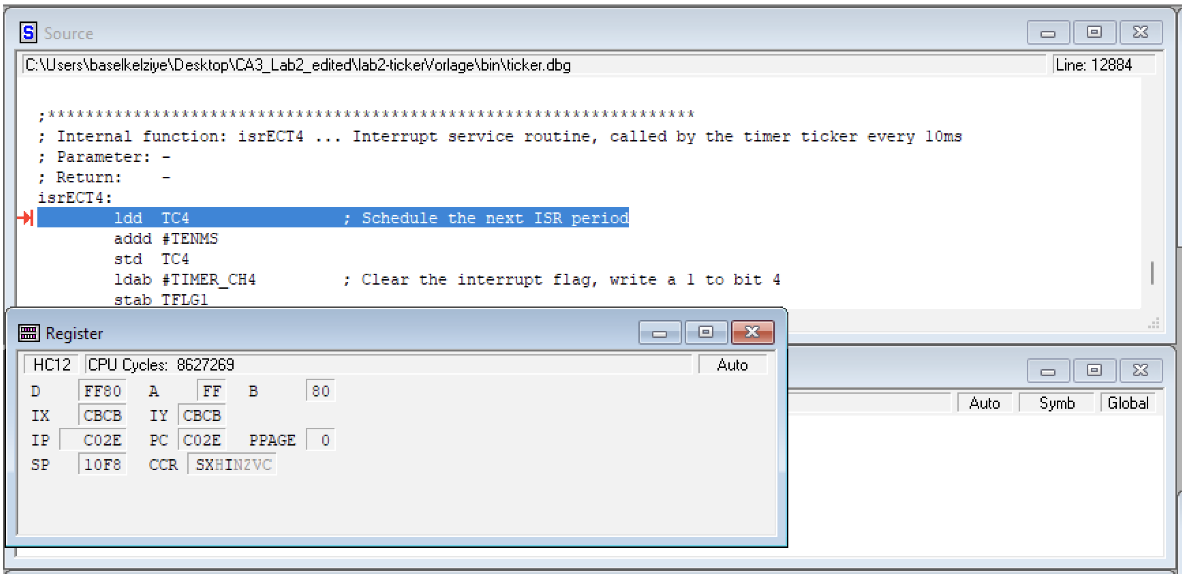


Figure 2: CPU Cycle After Second Call

as shown in Figure 2 our cpu cycle has increased to 8627269
to calculate the cycles between the 2 isrECT4 calls we do CYCLE2 - CYCLE1 = 8627269 - 8387252 = 240017
To calculate the elapsed time we can use the following formula:

$$ElapsedTime = \frac{Numberofcycles * 1000(ms)}{ClockFrequency}$$

And since our Dragon12 Board has a clock of 24MHz our equation look like:

$$ElapsedTime = \frac{240017 * 1000(ms)}{24 * 10^6} \approx 10ms$$

2 Uhr-voltage

2.1 buttons.c Module

Buttons are read using Polling, and only SW2-5 buttons are getting read. But while reading the buttons we need to Debounce so we don't trigger a function more than the intended amount read more.

Initilize buttons:

```
DDRH = 0x00;   //PORTH all input
```

So the method used to debounce is to read the button, save the state. Wait for a specified time and then re-read the button. If it matches the last state we can say that the button's signal is safe to read.

```
// Registers: Unchanged
void check_button(){

   unsigned char temp = PTH & PB_MASK; // only look at 2-5 switches only


   if(temp == (PTH & PB_MASK)){

   if(lastButton !=  temp){
   lastButton = temp;
   debounce = DEBOUNCE_DELAY; // wait before processing
   repeat = -1;  // signify initial depression
   repeatDelay = INITIAL_REPEAT_DELAY;
   return;
   }

    if (debounce != 0){ // we are debouncing
        debounce--;
        return;
    }


    if(temp == 4){ //SW2
    if(repeat > 0){
      repeat--;
    } else{
     repeat = repeatDelay;
     repeatDelay = REPEAT_DELAY;
     clockMode = ~clockMode;
    }
    }
```

Figure 3: Debouncing Buttons

in Figure 3 a part of the reading button function is given.
Corresponding on the clicked button a function is triggered.

2.2 Adding Time

To handle the overflows an intuitive way is to call the other adding function when overflows happen.

```
// Registers: Unchanged
void add_second(void){
  seconds_passed++;
  seconds = (seconds + 1)%60;
  if(seconds == 0){
    add_minute();
  }

}

// Public interface function: add_minute -> add 1 minutes
//called only from add_second, calls add_hour
// Parameter: -
// Return: -
// Registers: Unchanged
void add_minute(void){
  minutes = (minutes + 1)%60;
  if(minutes == 0){
    add_hour();
  }
}
```

Figure 4: Time Adding Functions

As an example if we have 60 seconds passed and we need to increment the minutes, instead of adding the minutes like minutes++ we call add_minute() function shown in Figure 4. And if we have an overflow in minutes now the add_minute() function will call add_hour() function and increments the hours. Using this strategy we get and clean and easy way to handle the overflows. Note: add_hour() function is implemented in the "Assembly-directive" Section to achieve the 12-Hours mode. (Will be covered later in the report).

2.3 clock.c Module

2.3.1 Print Names Periodically

Our print_names_periodically() function uses a global variable my_semaphore to decide which name to print. The value gets complemented when a change occurs.

```c
// Public interface function:print_names_periodically -> prints the name.  called every 10 seconds
//global variable my_semaphore changes state for the names
// Parameter: -
// Return: -
// Registers: Unchanged

void print_names_periodically(void){
  if(my_semaphore)
    {
    my_semaphore = 0;
    WriteLine_Wrapper("Basel@ITW2021/22",0);
    }
   else
   {
    my_semaphore = 1;
    WriteLine_Wrapper("Elias@ITW2021/22",0);
   }

}
```

Figure 5: Function to print the names periodically

2.3.2 Assembly Directives

Here add_hour and adjust_clock_string methods are generated recording to the SELECT12HOURS macro value.

2.3.2.1 add_hour

IF 12 Hours mode is selected our hours are getting modulus by 13 and a new global variable isPM is introduced to determine the "AM/PM" Time Zone.

```c
//Macro SELECT12HOURS if its set to 1
//add_hour function modulus on 13
//if hours = 12 isPM switches(from AM to PM vice-versa)
//if hours = 0 increment hours
#if  SELECT12HOURS == 1
void add_hour(void){
hours = (hours + 1)%13;
if(hours == 12){
  isPM =~isPM;
}

if(hours == 0){
  hours++;
 }
}
```

```c
#else
//its important to have the same name for the fu
//they function for 24H system (SELECT12HOURS =
void add_hour(void){
  hours = (hours + 1)%24;
}
```

Figure 6: Comparision of add_hour functions

In Figure 6. We can see the difference between the 2 compiled functions. If we use the 12Hour system(on the left) PM variable is used and we module by 13 and not 24.

2.3.2.2 adjust_clock_string

in Our adjust string functions the difference is that if we are in SELECT12HOURS mode (on the left) depending on isPM variable a "AM/PM" is added.

```c
void adjust_clock_string(void){
  result_char[0] = hour_char[4];
  result_char[1] = hour_char[5];
  result_char[2] = ':';
  result_char[3] = minutes_char[4];
  result_char[4] = minutes_char[5];
  result_char[5] = ':';
  result_char[6] = seconds_char[4];
  result_char[7] = seconds_char[5];
  result_char[9] = 'M';
  if(isPM){
    result_char[8] = 'P';
  } else{
   result_char[8] = 'A';
  }

}
```

```c
void adjust_clock_string(void){
  result_char[0] = hour_char[4];
  result_char[1] = hour_char[5];
  result_char[2] = ':';
  result_char[3] = minutes_char[4];
  result_char[4] = minutes_char[5];
  result_char[5] = ':';
  result_char[6] = seconds_char[4];
  result_char[7] = seconds_char[5];


}

#endif
```

Figure 7: Comparision of adjust_clock_string

2.3.3 Clock Pulse

This function is the heart of our program. It gets executed every second.

```c
void clock_pulse(void){
     add_second();
     decToASCII_Wrapper(hour_char,hours);
     decToASCII_Wrapper(minutes_char,minutes);
     decToASCII_Wrapper(seconds_char, seconds);
     adjust_clock_string();
     display_LED_C();
     if(seconds_passed%10 == 0){
      print_names_periodically();
      seconds_passed = 0;//prevent overflow
     }


     WriteLine_Wrapper(result_char,1);
}
```

Figure 8: clock_pulse function

as shown in Figure 8, first of all add_second() method is called, then we convert the hours,minutes and seconds to string using decToASCII function from LAB1. and then we create final string from the hours,minutes,seconds string. and we check if 10s is already passed to print names.

seconds_passed variable is incremented in add_seconds() method.

2.4 Thermometer.c

This module is written in C and provides methods to convert measured temperature values in a desired string format that is ready to be displayed on the LCD display. The two assembler routines initADC(void) and convertADC(void) from the adc.asm module are referenced and used here to control the data acquisition through the ADC. The integer variable always holds the latest temperature value in Degrees celsius ranging from -30°C to 70°C. The character Array holds the formatted textual representation that is fetched from other module with the getTemperatureText(void) function

2.4.1 initThermometer

is called once for initializing the the temperature variable with zero and to cofigure the ADC properly before measurements are taken.

2.4.2 updateThermometer

is called periodically in sync with the LCD display refresh that happens every second. It instructs the ADC module to convert and store the latest measurement.

2.5 Adc.asm

The module is written in Assembler and provides basic functionality to periodically read from the Analog-Digital-Converter using an interrupt.

2.5.1 initADC

routine enables the 10 bit ATD0 Analog-Digital-Converter of the dragon board and configures it to take 4 independent measurements before triggering the interrupt to signal a ¡¡ready for conversion¿¿ state.

2.5.2 adcISR

The interrrupt service routine adcISR is invoked as soon as the next 4 measurements have been taken. We sum them up and perform two logic right shifts to divide the sum by 4. This way we always have the average of 4 measurements which makes the system more robust. The average is stored and the interrupt flag resetted, so that the next measurements can be taken.

2.5.3 convertADC

The convertADC routine performs mapping of the raw value on to the degree celsius scale. It goes by the following formula:

$$\text{temperature} = \left(\frac{\text{adcVal} \times 100}{1023}\right) - 30$$

This routine is always called from the thermometer module right before the next temperature value is fetched.

2.6 main.c

in main i have display_led_C function which toggles the first LED0 every time it gets called, and if clockMode is in set mode it turn on the LED7.

```
void display_LED_C(void){  //displays the led corresponding to the clock's state

     PORTB = (PORTB ^ (1<<0));
      if(clockMode)
       {
        PORTB =  PORTB & 0x7F;
       } else{
        PORTB = PORTB | 0xFE;
       }


}
```

Figure 9: display_led() function

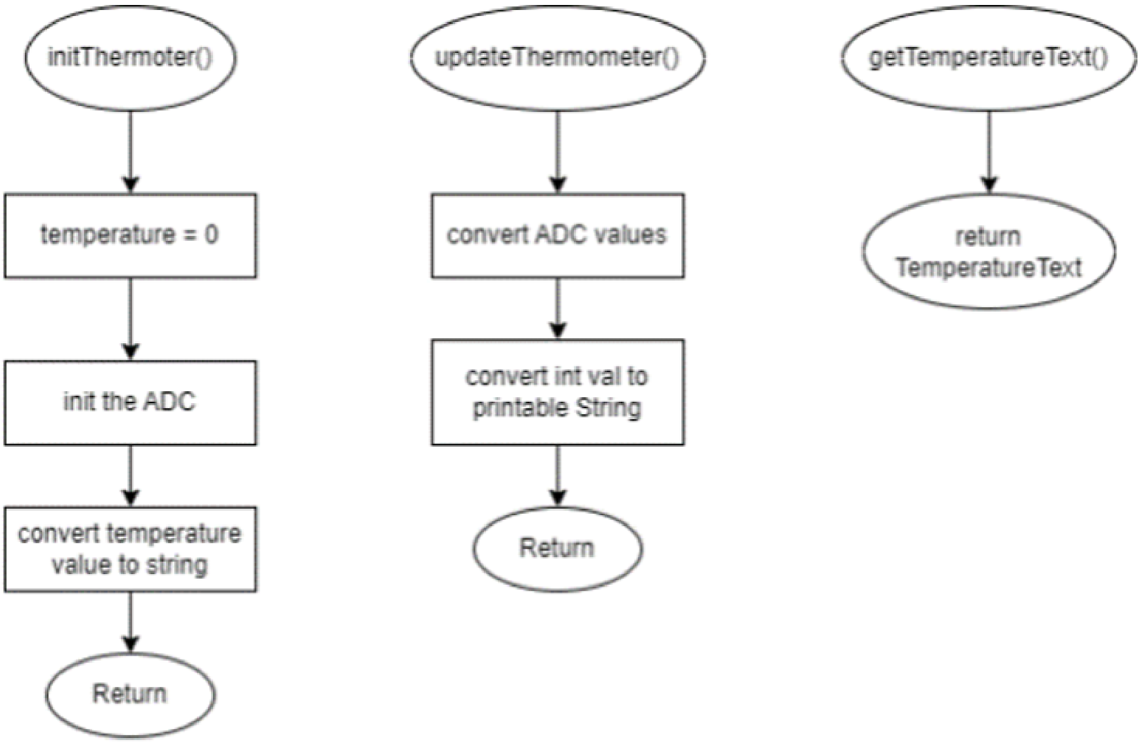
2.6.1 main function

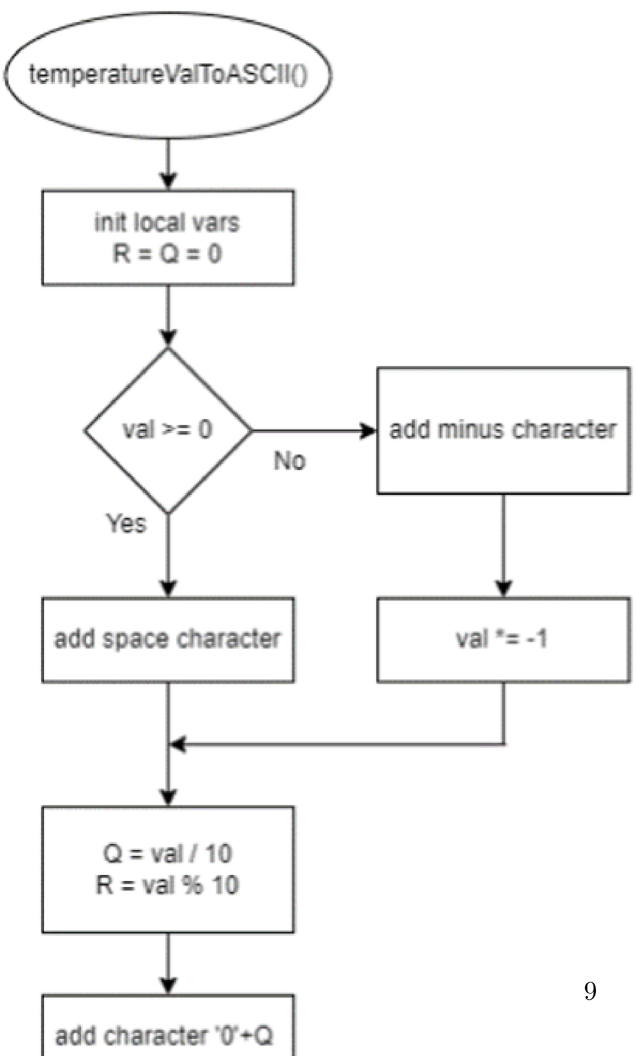
in main function we only initialize the hardware peripherals and call and endless loop. if clockEvent is truly value (non-zero) clock_pulse() method is called, and check_button() is called. the clockEvent value is changed every second in ticker.asm (Figure 10)

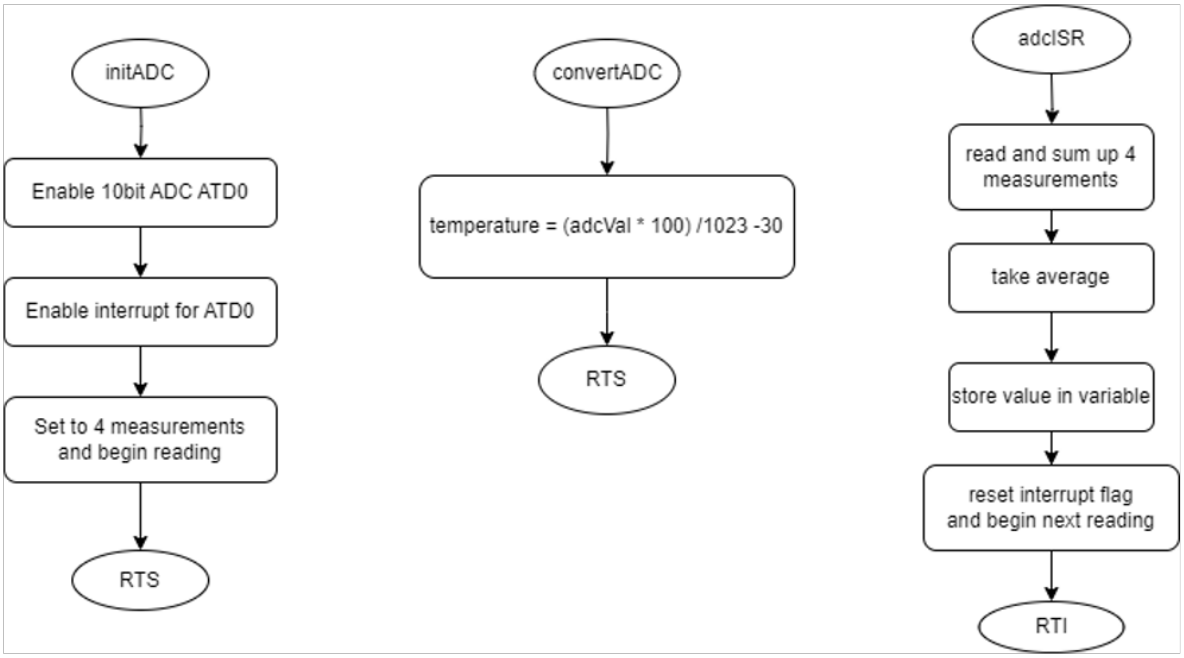
```
; --- Add user code here: Add whatever you want to do every second ---
        PSHB
          LDAB #1
          STAB clockEvent
        PULB


; --- End of user code ------------------------------------------------
```

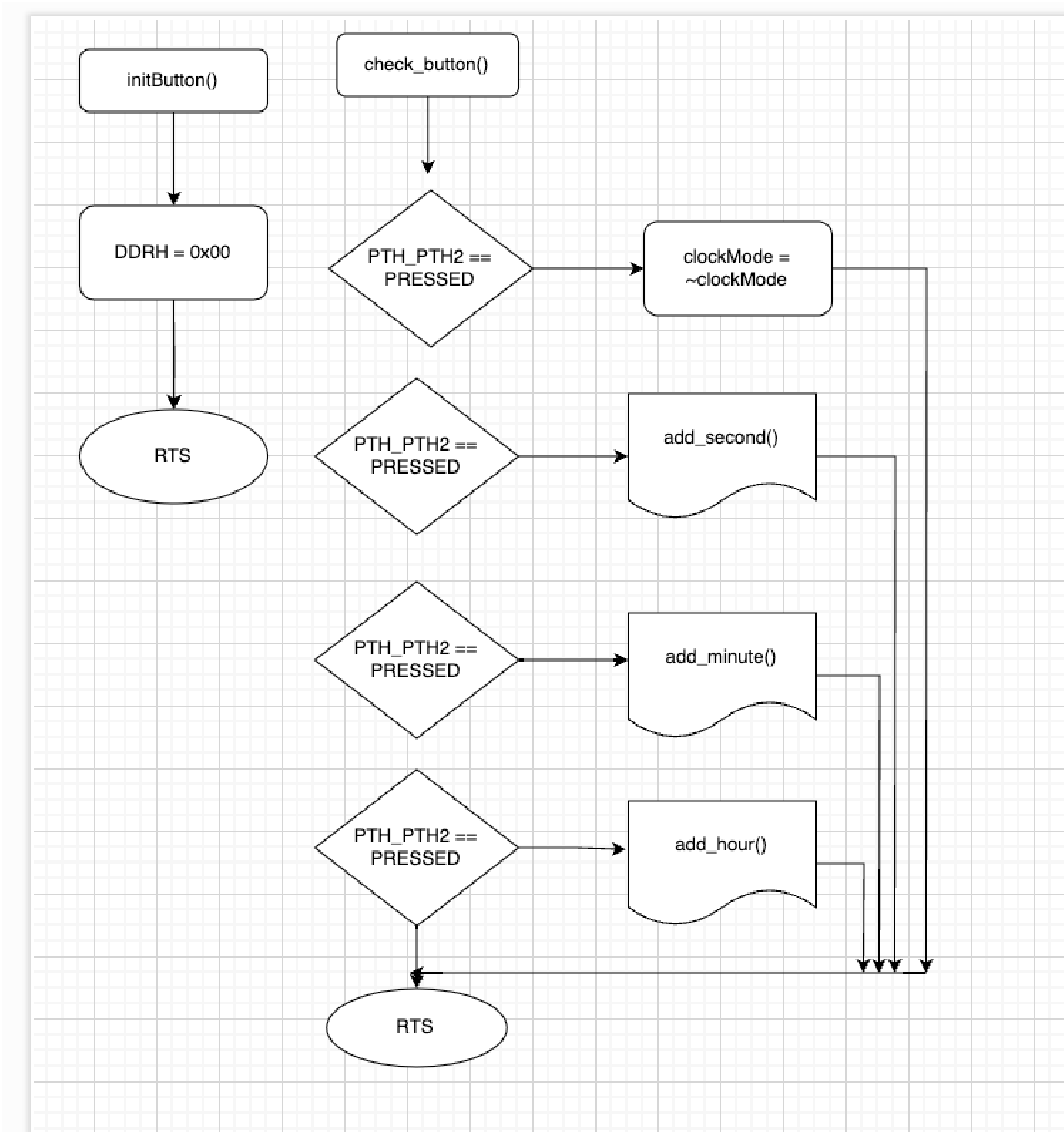
Figure 10: Changing clockMode in assembler

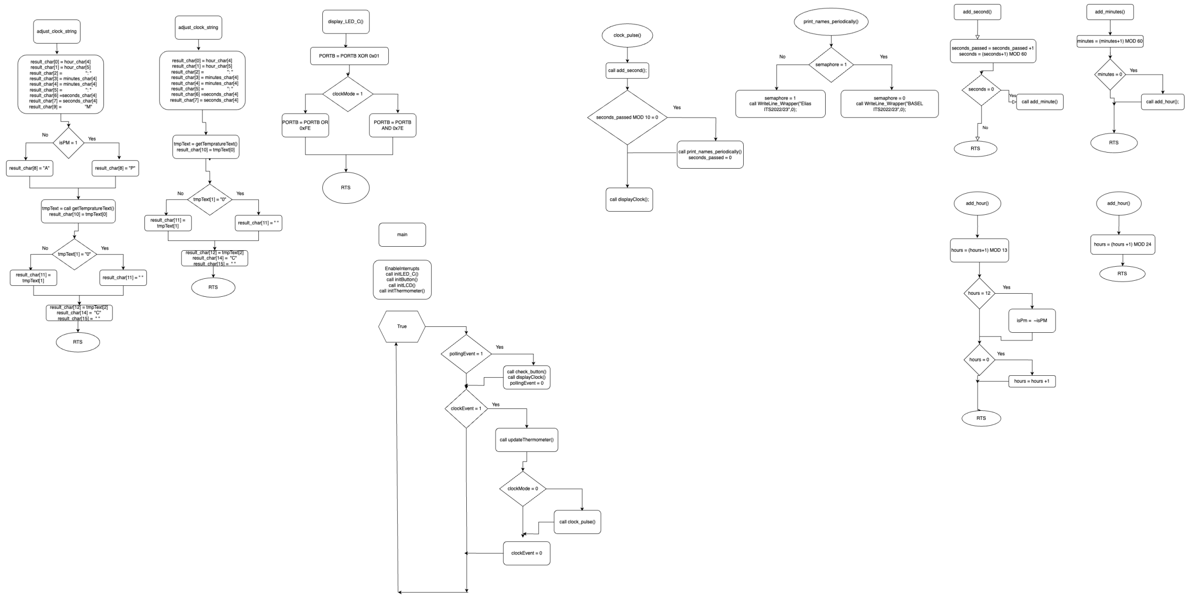
3 Flowcharts






initADC
Enable 10bit ADC ATD0
Enable interrupt for ATD0
Set to 4 measurements and begin reading
RTS
convertADC
temperature = (adcVal * 100) /1023 -30
RTS
adcISR
read and sum up 4 measurements
take average
store value in variable
reset interrupt flag and begin next reading
RTI


initButton()
DDRH = 0x00
RTS
check_button()
PTH_PTH2 == PRESSED
clockMode = ~clockMode
PTH_PTH2 == PRESSED
add_second()
PTH_PTH2 == PRESSED
add_minute()
PTH_PTH2 == PRESSED
add_hour()
RTS


adjust_clock_string
result_char[0] = hour_char[4]
result_char[1] = hour_char[5]
result_char[2] = ": "
result_char[3] = minutes_char[4]
result_char[4] = minutes_char[4]
result_char[5] = ": "
result_char[6] =seconds_char[4]
result_char[7] = seconds_char[4]
result_char[9] = "M"
isPM = 1
No
Yes
result_char[8] = "A"
result_char[8] = "P"
tmpText = call getTempratureText()
result_char[10] = tmpText[0]
tmpText[1] = "0"
result_char[11] = tmpText[1]
result_char[11] = " "
result_char[12] = tmpText[2]
result_char[14] = "C"
result_char[15] = " "
RTS
adjust_clock_string
tmpText = getTempratureText()
display_LED_C()
PORTB = PORTB XOR 0x01
clockMode = 1
PORTB = PORTB OR 0xFE
PORTB = PORTB AND 0x7E
main
EnableInterrupts
call initLED_C()
call initButton()
call initLCD()
call initThermometer()
True
pollingEvent = 1
call check_button()
call displayClock()
pollingEvent = 0
clockEvent = 1
call updateThermometer()
clockMode = 0
call clock_pulse()
clockEvent = 0
clock_pulse()
call add_second();
seconds_passed MOD 10 = 0
call print_names_periodically()
seconds_passed = 0
call displayClock();
print_names_periodically()
semaphore = 1
semaphore = 1
call WriteLine_Wrapper("Elias ITS2022/23",0);
semaphore = 0
call WriteLine_Wrapper("BASEL ITS2022/23",0);
add_second()
seconds_passed = seconds_passed +1
seconds = (seconds+1) MOD 60
seconds = 0
call add_minute()
add_minutes()
minutes = (minutes+1) MOD 60
minutes = 0
call add_hour();
add_hour()
hours = (hours+1) MOD 13
hours = 12
isPm = ~isPM
hours = 0
hours = hours +1
add_hour()
hours = (hours +1) MOD 24