Networking the Physical World: Lab 1

Samir Farhat Dominguez

March 21, 2022

Program

Code

```
#include "fsl_device_registers.h"
#include "fsl_debug_console.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "board.h"
#include <string.h>
#include <stdio.h>
#include <math.h>

#define SOURCE_CLOCK CLOCK_GetFreq(kCLOCK_CoreSysClk)
volatile uint32_t g_systickCounter;

void convert(int sec)
{
    char und_hr[50] = "0";

    int hr = floor(sec/3600);
    char hour[50];
    sprintf(hour, "%d", hr);
    if(strlen(hour) < 2) PRINTF(strcat(und_hr, hour));
    else PRINTF(hour);
    PRINTF(":");

    char und_min[50] = "0";
    sec = sec - hr * 3600;
    int min = floor(sec/60);
    char minute[50];
    sprintf(minute, "%d", min);
    if(strlen(minute) < 2) PRINTF(strcat(und_min, minute));
    else PRINTF(minute);
    PRINTF(":");
```

```
    char und_sec[50] = "0";
    sec = sec - min * 60;
    char second[50];
    sprintf(second, "%d", sec);
    if(strlen(second) < 2) PRINTF(strcat(und_sec, second));
    else PRINTF(second);

}


void SysTick_Handler(void)
{
    if (g_systickCounter != 0U) g_systickCounter--;
}

void SysTick_DelayTicks(uint32_t n)
{
    g_systickCounter = n;
    while (g_systickCounter != 0U) {}
}

int main(void)
{
    char ch;

    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitDebugConsole();

    if (SysTick_Config(SystemCoreClock / 1000U))
    {
        while (1) {}
    }

    int count = 0;
    while (1)
    {
        if(count%5 == 0) {
            convert(count);
            PRINTF("\n");
        }
        SysTick_DelayTicks(1000U);
        count++;
    }
}
```

Result

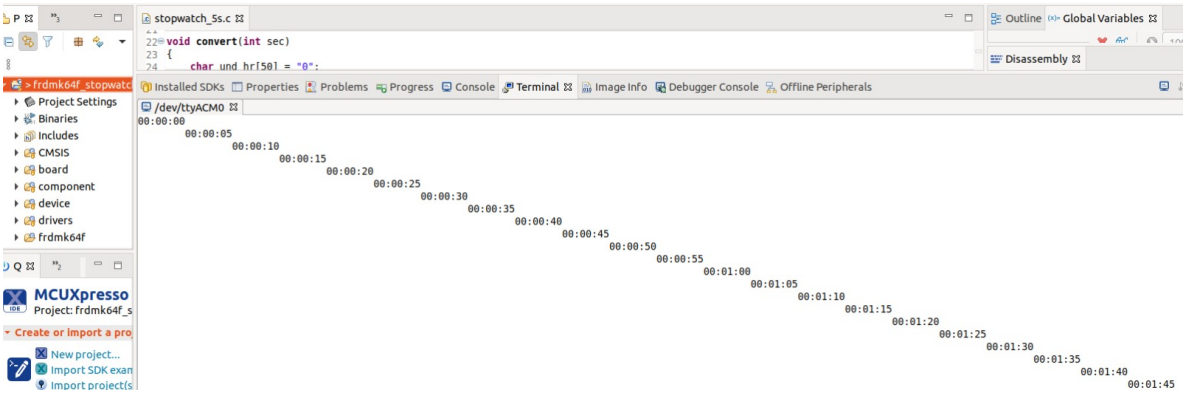

Figure 1: Output stepwise due to Ubuntu ACM0 USB Port Glitch

Discussion

AS far as the use of `PRINTF`, on Ubuntu the output is redirected to the `/dev/ttyACM0` port. This makes sense as it coincides with the USB port that the FRDM-K64 is connected through. By opening a virtual console in the MCUXpresso IDE at that port we can attain the redirected `PRINTF` output. We then utilize the statements in a similar manner to how it would be done in any other `C/C++` program. We ran into a common bug in the Ubuntu version of the IDE where the terminal outputs as a staircase, for the purposes of this assignment this is not a huge deal but it does make the output look awkward as above.

In the specific execution, the `convert(int)` function performs calculations for each digit(hours, minutes, seconds), deciding if it needs to prepend a '0' depending on the length. This means that if say 8 minutes have passed, the output will properly output "HH:08:SS" instead of "HH:8:SS". This makes it such that technically the output will be very slightly delayed due to calculations being performed between hours and minutes, as well as minutes and seconds. However this delay is in the realm of nanoseconds, and is thus imperceptible to a human. Additionally, `PRINTF` always has an inherent delay. as the output needs to be calculated, redirected and outputted, so it is not instantaneous even if it seems that way to the human eye.

In terms of delays, the intital attempt utilized the following code to attempt to count CPU cycles.

```
static void delay(volatile uint32_t nof) {
    while (nof !=0) {
        _asm("NOP");
        nof--;
    }
}
```

This was insufficient, as entering the while loop and performing the `NOP` operation took several CPU cycles to calculate, resulting in the timer actually lasting more than double

the expected delay. To create the working delay function the LED blinky demo timer was utilized, where the atomic unit for time `U` is used. This in addition to the `SysTick` functions and the `SOURCE_CLOCK` definition to accurately build the delay. The while loop and function stack create some sort of unexpected delay, but this is imperceptible to a human.

What are your thoughts about using an ISR to perform this function?

An interrupt service routine could be used to perform this function, however it needs to occur in a very precise way to avoid common issues. If this function were to incorporate other calculations or mechanisms, having an ISR could lock out other calculations that are necessary to be performed. There could be a designated interrupt latency to ensure the interrupt happens in a timely manner, but this strategy would not be efficient for the type of task to be performed, since it is a simple time delay.

The other issue with an ISR would be the possibility of context switches. As has been discussed in class, interrupts cause the execution to go from user mode to kernel mode, which then passes along the context to the interrupt handler. Both these steps require steep overhead, not anything that a human can notice but the precision of the timer could be slightly affected. Additionally, having to run an interrupt that frequently can cause scheduling issues if other programs are trying to run on the same core as the stopwatch.

Despite all these factors, an ISR is actually the recommended way to build the application, and actually how timing takes place at the OS level in Unix systems. ISRs are used to calculate absolute time as well as elapsed time for programs. In fact, there is a dedicated kernel function which is called at each timer interrupt, where the timing registers are read and updated at every ISR cycle. The below figure depicts this

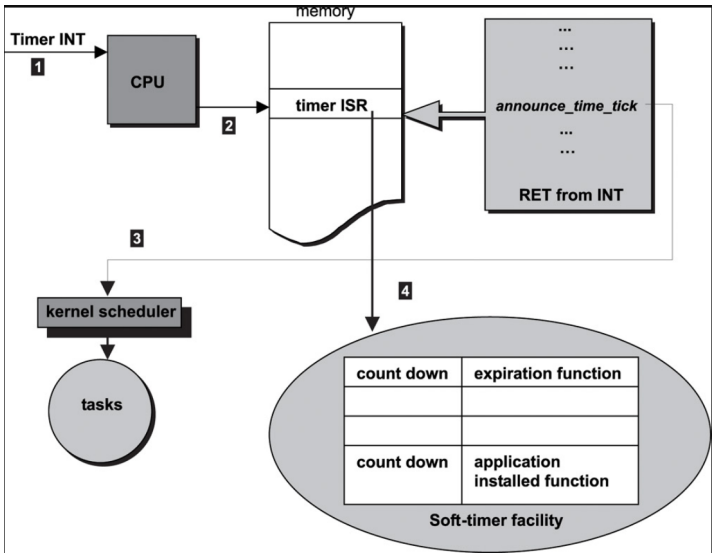

Figure 2: Embedded Linux ISR Blocks[1]

Raghavan, Pichai, Amol Lad, and Sriram Neelakandan. Embedded Linux system design and development. Auerbach Publications, 2005.