ELE 302 Clock Project Report

Chris Tralie and Carlton Chow

You documentation must include the assembly language listing of the program as well as a write-up describing the operation of your system. This must include definitions of each variable, a functional description of each major section of code, and an overall description of the mechanisms used to instantiate each important function of the system. Some of this can be included as comments in the original code listing. You should also include a measure of your minimum RAM and FLASH requirements and a derivation showing that you have met the accuracy requirement

I. High-Level Operational Overview

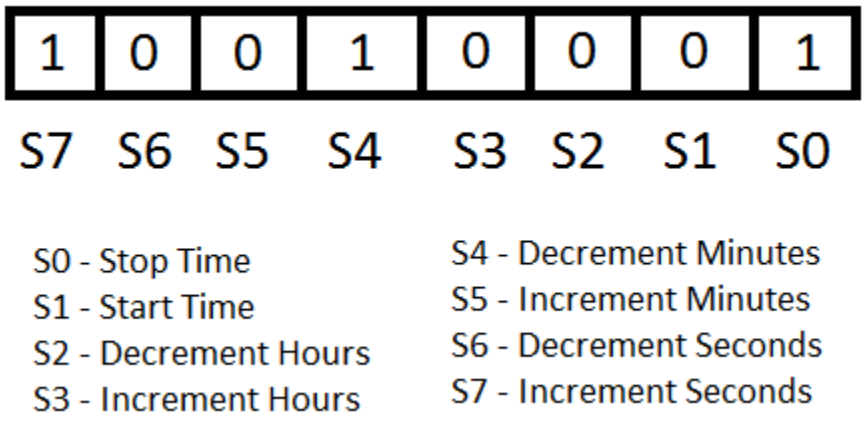
We chose to use 6 variables to store time (2 for each of seconds, minutes, and hours) + 1 variable for AM/PM. Though the code for telling how to increment/decrement time is more complicated than it would have been for a single variable for time, it makes updating the display a much more straightforward process.

We use a single quad timer with an interrupt as our basic model for counting. We calibrated the quad timer so that this interrupt is triggered every $1/100^{th}$ of a second (explained further in section IV). We then have a counter that increments each time this interrupt is triggered. If counting is enabled, the seconds will be incremented each time our centi-second counter reaches 100 (at which point our centi-second counter is reset). Regardless of whether timing is enabled, every $1/100^{th}$ of a second, the buttons are polled and appropriate action is taken to increment/decrement seconds/minutes/hours based on which button is pressed. NOTE: The interrupt is where all of the action takes place, so the main program is simply an infinite loop to allow the program to continue to run while the interrupt is waiting to be triggered by the timer.

The basic process for looking at the buttons is as follows:

*Since there is only one line that feeds back from the clock board to the microprocessor, each button needs to be polled individually. But specifying which of the 8 buttons to check uses the same 3 address lines through the GPIO device that are used to select which display to change. Therefore, the display gets wiped each time a button is checked in this manner. To deal with this, we simply update all of the BCD displays instantly after the button it checked to display the correct time value.

*The 8 buttons are polled one after another, and their current state is stored in a bitmask:



In this example, the "increment seconds," "decrement minutes," and "stop time" buttons are being pressed in a centi-second timeslice.

Once the button state bitmask has been updated at each step, a function is called that takes appropriate action. Buttons S2-S7 each have their own timing variable that counts how many centiseconds they have remained 1. If they have remained 1 for 0.5 seconds (50 centiseconds), then increment or decrement the time (depending on the button in question). If they remain pressed for an addition 0.25 seconds (up to 75 centiseconds), then increment/decrement again, subtract 0.24 seconds and continue this process so that if the user holds down, the time continues to increment/decrement every 0.24 seconds.

If the user presses one of the buttons S2-S7 but releases before 0.5 seconds, then implement debouncing functionality; that is, don't increment/decrement unless the button has been pressed for more than $1/20^{th}$ of a second. This still gives the user plenty of time to press/release the button, but it makes sure that extra presses don't get recorded.

II. Code Summary

a. Function Calling conventions: Instead of bothering ourselves with the stack, which, we tried to use a more intuitive approach: we defined three global variables arg1, arg2, and arg3 that can be used to pass arguments to functions, and we declared a variable ret1 to hold the result of a function.

b. Variables:

- int arg1, arg2, arg3; //Stores values that are passed to functions
- int ret1; //Stores the return value from a function
- int buttons; //The buttons bitmask for storing pressed states of buttons

- int seconds1, seconds2; //Seconds1 stores the least significant digit of seconds, seconds2 stores the most significant digit of seconds
- int minuets1, minutes2; //Same as seconds, but for minutes
- int hours1, hours2; //Same as seconds and minutes but for hours
- int AMPM; //0 if am, 1 if pm
- int sadvtimer, sdectimer, madvtimer, mdectimer, hadvtimer, hdectimer; //The centi-second timers for the different buttons
- int counting; //1 if time is advancing, 0 if time is frozen
- int centicounter; //Counts the how many 1/100 second intervals have passed

c. Function Descriptions:

- initGPIO: This function sets up pins 0-7 of the GPIO to be output to the clock board, and pin 8 to be a feedback input
- clearAllDisplays: This function writes the value 1111 to all eight displays. This is not a valid number between 0-9, so the hardware will clear everything on the displays
- displayAllEights: This function writes 1000 to all eight displays to check to make sure everything is lighting up properly (we had some soldering issues towards the beginning that were causing displays to flake out)
- lightDisplay: Accepts 2 argument: 1st argument selects which display (0-6), second argument says what value to send it (0-9). This writes the appropriate values to the output part of GPIO
- lightAMPM: Select the "display" 110 (the JK flip flop for controlling the AM/PM) and write 0010 to it to select AM, and 0001 to select PM LED. NOTE: For some reason (probably to do with timing) we had to insert a junk GPIO command before actually trying to update the flip-flop. To do this, we just wrote 0000 to "display" 111 (which does nothing), a hack that never failed to solve this problem.
- updateButtons: Poll all 8 buttons and update the button bitmask (by doing lots of bit-shifting). Update the all displays afterwards since polling clears them
- getButtonState: Used to extract the state of a button (1 or 0) from the bitmask; a convenient subroutine to call to save code in other parts of the program

- doButtonFunctions: Increment each of the button centisecond timers, and take appropriate action based on how long they have been held down (as discussed in section I).

- advanceTime: Accept one argument (0-seconds, 1-minutes, 2-hours), and increments the chosen time component. Takes care of seconds overflowing to minutes, minutes overflowing to hours, and changes of hours from 11-12 or 12-11 toggling the AM/PM. This function is called either when 100 centiseconds have been counted and counting is enabled (in which case the argument is always 0), or when a button is pressed (where the argument can be seconds, minutes, or hours based on what the user wants).

- decrementTime: Does the same thing as advanceTime, but in reverse. This function can only be called during an interrupt if a button is pressed for a certain amount of time or released, since time never goes backwards when it's counting naturally

- displayTime: Refreshes all displays to display the appropriate variables: seconds1, seconds2, minutes1, minutes2, hours1, hours2, and AMPM (Note that the state of AMPM is stored explicitly and refreshed every time; we don't rely on the toggling feature of the AM/PM).

- TimerInterrupt: Called every 1/100th of a second counted by one of the quad timers. Updates all of the sates of the buttons (calls updateButtons) and checks to see if the buttons trigger any actions (calls doButtonFunctions). It then increments centiseconds and increments seconds if 100 centiseconds have elapsed and timing is enabled. Regardless, the centisecond timer is always reset to 0 at the end of 100 centiseconds to prevent overflow. Also note that since the buttons are checked before the 100 centisecond comparison is done, there's only one call to advanceTime / decrementTime going on at once sequentially. This ensures that the logic is correct

III. Memory Usage

*The program takes up 4182 bytes on the board's flash memory

*There are 20 global variables, 2 bytes each. There are at most 3 callee-saved 2-byte registers in a subroutine. Disregarding whatever else is on the stack frame, we are responsible for (23*2) = **56 bytes** of RAM usage

IV. Timing Calibration

We chose to use the IPBUS clock along with a prescaler to drive our counter. Initially we set our prescaler to a factor of 128 so that the counter incremented every 128 cycles of the processor. We set our counter to go up to an adjustable value before calling an interrupt and resetting back to zero. The limiting value which we wanted to set was supposed to match up with the counter reaching this value at 1/100 of a second. At first by trial and error, we saw that 2007 seemed to be a good approximation. We left the clock running for 24 hours, and then saw that ours was fast by about a minute and a half. Since the clock was fast, we needed to increase the number of cycles that the counter would run through before calling the interrupt to notify our program that 1/100 of a second had passed. We then increased the counter limit to 2008 (still with a 128 factor prescaler); we waited for 12 hours and saw that our clock was still fast by approximately 20 seconds, which meant that it would be fast by about 40 seconds in 24 hours. We realized that incrementing the counter limit once more would probably result in our clock being too slow, so we changed our prescaler to a factor of 64, so we multiplied our counter limit by two, and then added one to slow it down slightly for a new counter limit of 4017. After another 24 hours our clock was slightly slower than a normal clock, by about 10 seconds, so we left our prescaler and counter limit at these values.