**Chronometer using big main loop technique**

Luis Raul Villanueva Lopez (raulvlopez@gmail.com)

Iram Castañeda ([blinketito@gmail.com](mailto:blinketito@gmail.com))

May the 20th, 2013

**Universidad Autónoma de Guadalajara**

**Abstract**

There are many ways to control the use of microprocessor time in a way that we are also able to control at what time we want multiple tasks to be consumed by the microprocessor. Generally speaking, almost every known microcontroller support timer interrupts as a way to control at what time certain tasks are to be executed. Notwithstanding, there is also another approach to accomplish almost the same without using timer interrupts. This technique is known as the big main loop and we will discuss the development of a chronometer that uses that technique.

**Introduction**

It is well known that microprocessors support interrupts to the normal execution indicating an event that requires immediate action. These interrupts may be generated by either internal hardware, i.e.: timer interrupts, I2C interrupts, GPIO interrupts among others.

Yet, there are applications where the use of microcontrollers with no interrupts is our only choice, for example, due to the low cost of these devices. If that is the case, the big main loop technique may be very useful to handle applications where different tasks shall be executed at different intervals.

Now, the big main loop technique basically consists of a loop that takes a fixed amount of cpu cycles in a form of a delay. As this delay will be our reference time for all of our tasks, it is desired that all tasks intervals are multiple of our reference time in order to have more accurate timings. Thus, we first need to calculate and decide the Greatest Common Divisor (GCD) of all tasks. Then, all task intervals will be calculated based on the GCD as reference. The rest is just a matter of counting down until we meet the required time for each particular task.

**Procedures**

Our microcontroller in our evaluation board runs natively at 16Mhz with the existing configuration. Also, our delay CPU instructions, takes approximately 12 cycles to complete for every loop. Our GCD for all of our tasks turned out to be 500E-6 which will become our time base for all other tasks.

Dividing our time base by our delay loop period gives the number of cycles that our delay loop has to execute each time.

This is shown in red below:

|  |  |
| --- | --- |
| **Description** | **Calculation** |
| CPU Clock | 1.60E+07 |
| CPU cycle duration | 6.25E-08 |
| Delay Cyles | 12 |
| Delay Period | 7.50E-07 |
| Delay Loop Period | 666.7E+0 |

Now, every task will have to implement its own counter to keep track of its own period. For example, one of our LED indicators hast to be toggling every 250 milliseconds, then, we will need a counter for this task that will be executed every 500 cycles of our time base to meet that condition. Full table with is shown below:

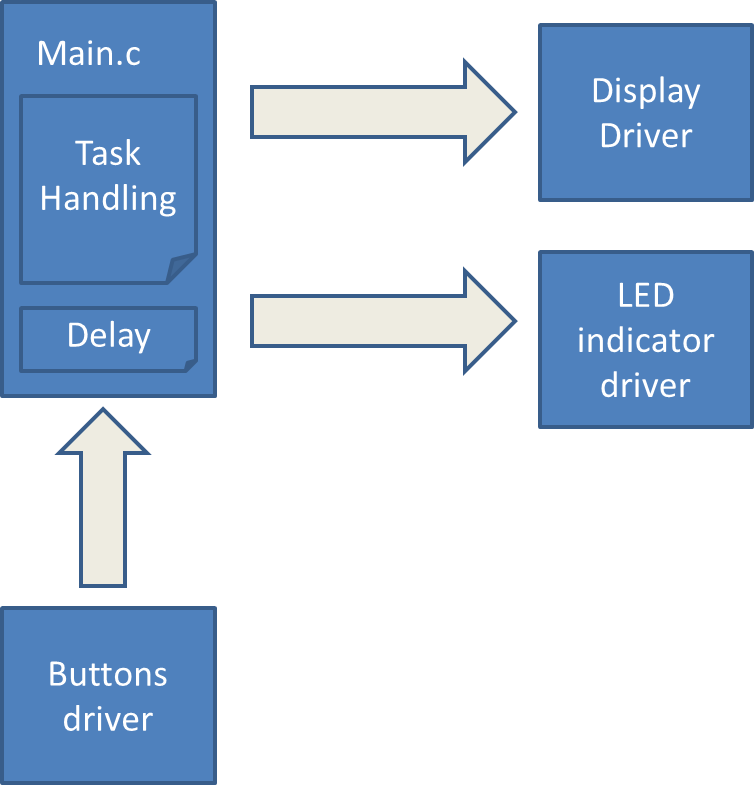
|  |  |  |  |
| --- | --- | --- | --- |
| GreatestCommonDiv | 500.0E-6 |  |  |
| **Task** | **Period** | **Freq** | **Count** |
| Displays | 3.5E-3 | 285.71 | 7 |
| Buttons | 50.0E-3 | 20.00 | 100 |
| Time | 100.0E-3 | 10.00 | 200 |
| LED2 | 166.0E-3 | 6.02 | 332 |
| LED1 | 250.0E-3 | 4.00 | 500 |

Once that is done, implementation is easy, as the program has to handle separate counters for every task such as every time the counter expires, the task is executed. The content of every task is up to the developer, but in this case we separate our application into 4 main areas:

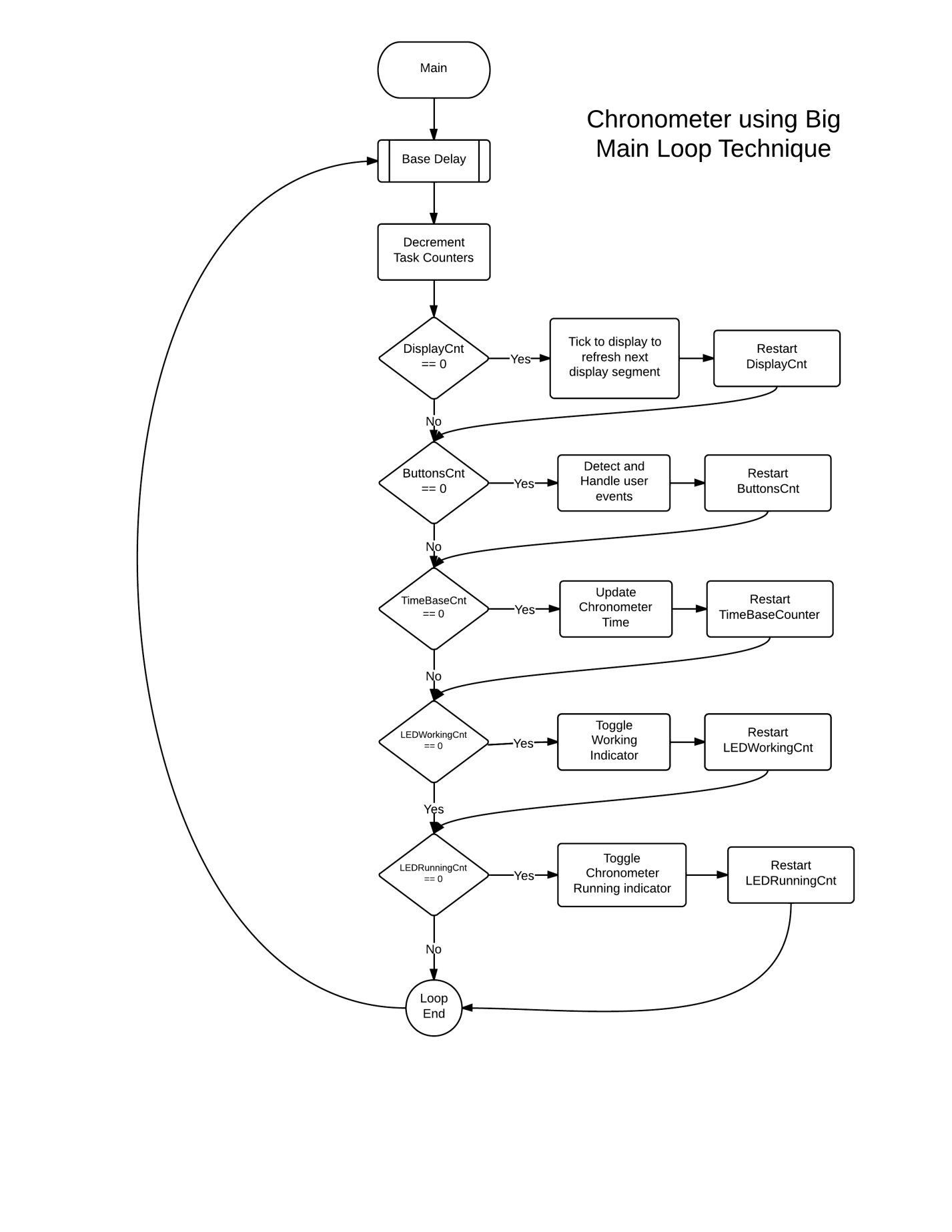
* The display task
* The buttons task
* The time handler (chronometer) task
* And the LED indicators.

In order to simplify things, we created device drivers for each of these so that in our main.c we should have only the necessary calls and code is cleaner.

This is the Block Diagram:



And its corresponding summarized flow diagram:



Last but not least, here is the math to calculate an approximate of the CPU time utilized by all tasks for our chronometer:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Frec | 16.0E+6 |  | Cycles Per Instruction | 2 |  |
|  | **Task** | **Instructions** | **TimesPerSecond** | **Duration** | **Total Duration** |
|  | Display | 35 | 280 | 4.4E-6 | 1.2E-3 |
|  | Buttons | 20 | 50 | 2.5E-6 | 125.0E-6 |
|  | LED1 | 10 | 4 | 1.3E-6 | 5.0E-6 |
|  | LED2 | 20 | 6 | 2.5E-6 | 15.0E-6 |
|  | Time | 50 | 10 | 6.3E-6 | 62.5E-6 |
|  |  |  |  | **Total** | **1.4E-3** |

As you can see, we are actually using about 2 milliseconds of CPU time which is roughly 0.2%.

**Conclusion**

Even though I have my doubts on the accuracy of the big main loop technique, the chronometer was finally complete. At first glance, the timings for every seconds seems very accurate. I could not check with an oscilloscope to understand the error rate of timings.

I got many obstacles while developing this application, such as the following:

* Lack of practice with hardware became in many troubles while assembling our prototype. Being a couple of years in the software industry with no direct access to hardware was an obstacle. So I had to relearn.
* Open drain, open collector and that kind of GPIO configurations were confused at first as I tried many times and could not have the GPIOs working. It turned out it was not a FW but a HW problem.
* It was unknown to me that the GPIO ports have to be clock gated first in order to start using them, which becomes very convenient for low power consumption applications but got me stuck for a while until figured out.
* Even though the TI Stellaris Launchpad has their own libraries to handle almost everything in the evaluation board, I suffered even more trying to use them than creating my own drivers with raw register access. Kind of weird since they are made to make things simpler.

On the other side, I really enjoyed this practice it was a while since the last time I developed FW from scratch. Usually at work, one only develops things at a certain level (new features/maintenance) and no chance to create real life applications. This project really improved my embedded development skills.

Also, it is interesting that even for a 12 USD evaluation board, our application is barely consuming 0.2 % of its CPU time. I really wonder what kind of applications may come to utilize full CPU time.