mp3: Timing Interrupts

Assigned: 14 November 2017 **Due:** 28 November 2017 class time

I. OBJECTIVES:

The purpose of this assignment is to build a package of timing routines for the tutor VM and use it to time code on the tutor VM. This code does not run on UNIX. Read the Intel 8254 Programmable Interval Timer (PIT) Data Sheet. Look at $pcex/timer.c for an example of C code for accessing the timer. Copy all the files needed for this project from /courses/cs341/f17/cheungr/mp4/.

II. The timing package:

A package is a set of utilities that can be called by a program that wants to use them. When designing a package we carefully consider the things in it that must be visible to the caller, specify those, and require that everything else be internal and invisible. The visible parts of the timing package you will build are in "timepack.h" (function prototypes and comments are listed in the timing directory /groups/ulab/pcdev/timing). A customer for the package's services includes that header file (using #include) in the program for compilation and links his or her object code with "timepack_sapc.opc" to build the executable. You are asked to modify "timepack_sapc.c" to provide a higher resolution timer service for programs running on the tutor VM.

a) Part 1: Existing Code

Every package should have a test program showing that it works by calling it. This test program is called a "driver" because it sits on top of the package and drives it like we test-drive a car – start up, do this, do that, stop, shut down. It is also called a "unit test" program because it tests just this one package separate from any other package in a bigger program. If you suspect something is wrong in a certain package, you'll try to make its unit test fail, and then you debug the problem in the relatively simple environment of the unit test, rather than in the bigger program.

The test program for timepack is testpack.c. You can build testpack and testpack.lnx right away and run them. Note what is printed out by testpack.lnx. Capture this in a typescript1 file. It shows that the timing package (as provided) can time things on the tutor VM to 55-ms accuracy, but not to the microsecond accuracy we want. The next step is to get your timepack_sapc.c fixed up for the higher resolution and the unit test executable testpack.lnx will show it.

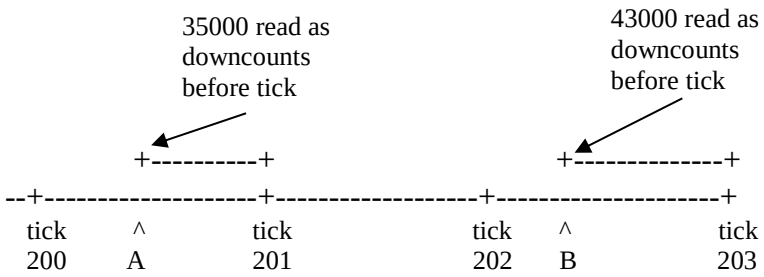
b) Part 2: Modifed Code:

You're working directly with the hardware device, the Programmable Interval Timer (PIT) with its interrupt handler. This code has been provided to you in a). The timepack_sapc.c as provided can measure time in timer "ticks" at an 18.2-Hz (55 ms/tick) standard PC tick rate.To use the PIT to measure higher precision, you make use of "downcounts" within the timer chip. What you need to do is determine how many counts have downcounted in the

timer *since* the last tick and compute a higher accuracy time. By doing this at both the start and the end of the time interval being measured, you can compute the elapsed time accurate to a few microseconds, a very respectable timer service. You'll need to modify the timepack_sapc.c to achieve this.

There are 64K = 64*1024 downcounts within 1 tick and the time for one downcount to take place is _____usec.You will use this value later for your calculations.

Example: Event A happens 35000 downcounts before tick 201
Event B happens 43000 downcounts before tick 203



Since timer downcounts count down from 64K at the tick, you need to subtract the register value from 65536 to get the number of downcounts since last tick:

number of downcounts since tick = 65536 - observed_count (in register)

Thus the accurate time between A and B is
= 202 clock ticks + (65536 – 43000) downcounts
- (200 clock ticks + (65536 – 35000) downcounts)
= 2 clock ticks – 8000 downcounts
where 1 tick = 64*1024 downcounts, so this whole thing can be expressed in downcounts, and then converted to usecs.

Note that you need to *convert* downcounts (an int) to usecs (another int). To do this, cast the downcount value to double, multiply by the correct double conversion factor, and then cast back to int usecs. The idea here is that the conversion factor is itself non-integral, so the needed multiplication by it needs to be in floating point, resulting in a floating point number accurate to about 1 usec, so we might as well cast it back to an int value since ints are easier to work with.

See the timer.c in the course directory for a C program that reads the timer downcounts – you can take code from this as needed.

Don't leave any printf's in your final code that output during timing runs!! They take a really long time and ruin the data. Save the timing figures in memory and print them out after the stoptimer call.

Any private functions or variables you need should be declared static in "timepack_sapc.c", so that the calling program can never accidentally use a variable or function with the same name, or interfere with yours. Capture your improved result in a typescript2 file.

FINAL NOTE:

In the event that you are unable to correctly complete this assignment by the due date, do not remove the work you were able to accomplish – partial credit is always better than none.