ECE 362 Lab Verification / Evaluation Form

Experiment 4

Evaluation:

IMPORTANT! You must complete this experiment during your scheduled lab period. All work for this experiment must be demonstrated to and verified by your lab instructor *before the end* of your scheduled lab period.

STEP	DESCRIPTION	MAX	SCORE
1	`delay`	4	
2	`clock`	8	
3	`tdisp`	4	
4	`prompt`	6	
5	`Thought Questions`	3	
	TOTAL	25	

Signature of Evaluator: _____

Academic Honesty Statement:

IMPORTANT! Please carefully read and sign the Academic Honesty Statement, below. *You will not receive credit for this lab experiment unless this statement is signed in the presence of your lab instructor.*

"In signing this statement, I hereby certify that the work on this homework and experiment is my own and that I have not copied the work of any other student (past or present) while completing them. I understand that if I fail to honor this agreement, I will receive a score of ZERO and be subject to possible disciplinary action."

Printed Name: _____ Class No. __ __ __ __ - __

Signature: _____ Date: _____

Experiment 4: "Tick Tock" Clock Based on Software Delay

Instructional Objectives:

- To learn how to write a structured program
- To learn what is meant by top-down design
- To learn how to break a programming task into modules and to learn the advantages of such an approach
- To enhance your ability to write, test, and debug HC(S)12 assembly code

Prelab Preparation:

- Read this document in its entirety
- Bring your own 9S12C32 board with you to lab – target systems will not be provided
- Write the code for *all* steps prior to coming to lab

Introduction:

As the jobs we desire the 9S12C32 to do become larger, it becomes more and more complicated to write and debug the code. One technique for overcoming these complications (or at least simplifying the problem) is *modular programming*. The idea behind modular programming is to break the job into small tasks and, if needed, break these into even smaller sub-tasks. The basic premise is that smaller tasks are easier to code, test, and debug.

The toughest part of modularizing your code is often deciding *how* to break it into appropriately-sized modules. The technique we propose you use is *top-down programming*. "Top-down" simply means start at the "top" (the "big picture") and work your way "down" to the bottom (the "program pieces"). In other words, first write a "main" (or "supervisor") code module that calls various sub-modules. Each of these sub-modules performs a specific task. Each task may be quite independent of the others. This makes it easy to program each one separately. It may be helpful to even break these sub-modules into several lower-level "*chunks*". Each "chunk" will perform a single job and will probably not exceed much more than 10 to 15 lines of code. These chunks will perform the elementary tasks needed by the "main" program. For example, a "code chunk" may include reading a value from the terminal keyboard or converting an ASCII value to a hex value. The important thing to note is that most all of these "code chunks" can be *independently tested, debugged, and verified*. As you test and verify various "code chunks", you can successively incorporate these modules into your main program (note that "include" files can be utilized to help facilitate this process). For routines not yet written, simply write a "dummy module" that merely returns or prints a message to the effect that the module has been reached. If you wish execution to terminate when one of these yet-to-be-completed sub-modules is reached, simply set a breakpoint in that "dummy module".

In this experiment, you will create a (12 hour) time-of-day clock that executes out of flash memory upon power-on/reset. The timing reference for this clock will be based strictly on software delay loops. Once your clock is working, you will analyze the accuracy of your solution.

Program Description:
For this experiment, you will create a 12-hour time-of-day clock that updates its display every second. The clock application should be loaded into flash memory and run upon reset (i.e., it should function in a "turn key" fashion). Upon startup/reset, the user should be prompted for the initial time setting as illustrated in the dialog box below; each user entry should be checked for validity (i.e., "legal" time, only "a" or "p" for am/pm, etc.). The application should keep running until the processor is reset.

```
9S12C32 Tick Tock Clock V1.0
Created by: your name and class number
Last updated: date code was last updated

Enter hours: 00
*** ERROR *** Invalid entry - try again
Enter hours: 01
Enter minutes: A
*** ERROR *** Invalid entry - try again
Enter minutes: 30
Enter seconds: 99
*** ERROR *** Invalid entry - try again
Enter seconds: 2B
*** ERROR *** Invalid entry - try again
Enter seconds: 15
Enter a/p: x
*** ERROR *** Invalid entry - try again
Enter a/p: p
Starting...
01:30:15 pm
01:30:16 pm
01:30:17 pm
.
.
.
```

Download and unzip the Code Warrior project folder **Lab4** from the course website. Note that the project is set up to communicate with the target system (9S12C32 microcontroller board) using a **USBDMLT** pod (hence the selection of **TBDML** mode). The serial port on the 9S12C32 board is set up to communicate with the host PC using a standard serial-to-USB adapter, in conjunction with the Windows application **TeraTerm** (or most any other terminal emulator program).

Note: Be very careful when connecting your board to the **USBDMLT** pod (6-pin header) – in particular, carefully note the location of "pin 1" and ***do not pull on the cable*** when removing the header. The **UBDMLT** pods in lab are set up to power your board directly, so no external power supply (wall wart) is necessary.

Follow the steps outlined below in creating your solution.

Step 1:
Examine `main.asm` and note the routines provided as well as the ones you will need to write. Also, examine the flash memory "boot up" code, and note that the processor will be running at 24 MHz (which means each machine instruction cycle is 41.67 ns). Finally, note that variable declarations provided (others may be added, but all variables must be in SRAM).

Write and test a `delay` subroutine that provides a variable delay (up to 1000 ms) based on the value based to it via the X register. This can be accomplished using a doubly-nested loop similar to the one used in the lecture notes, except note the extended "outer loop" range as well as the difference in machine instruction cycle time (41.67 ns) based on a CPU clock speed of 24 MHz. Test your `delay` subroutine by using the `showcycles` command in Code Warrior to test its accuracy for different values passed in (X).

In the space provided below (or on an attached sheet), sketch a flowchart for your `delay` subroutine.

Step 2:
Write and test a `clock` subroutine that increments the current time by one each time it is called. Note that each time-related variable (`hours`, `minutes`, `seconds`) is stored as a packed BCD number, and that the valid range for `minutes` and `seconds` is `00` to `59` while the valid range for `hours` is `01` to `12`. Also note that your `clock` routine should "toggle" the am/pm flag (`ampm`) when `hours` changes from `11` to `12` (a *flag* is a single byte variable that is either 0 or 1). Test your clock subroutine code by initializing the time variables using the debugger and setting up a simple loop that calls `clock` to test its operation.

In the space provided below (or on an attached sheet), sketch a flowchart for your `clock` routine.

Step 3:
Write and test a `tdisp` subroutine that displays the current time in **`nn:nn:nn a/p`** format on a new line of an emulated terminal each time it is called. Note that the subroutines `disbyte`, `htoa`, and `outchar` are provided to assist with this task. To test your `tdisp` subroutine, simply create an "infinite" loop that calls `tdisp`, `clock`, and `delay` on each iteration (which will essentially become the heart of your "main" program).

Step 4:
Write and test a `prompt` routine that prompts the user for the initial time setting and checks the validity of the user-entered responses as they are entered (see dialog box under the Program Description section for an illustration of how it is to work). Note that the subroutines `atoh`, `inchar`, and `pmsg` along with a `print` macro are provided to assist with this task. Once your `prompt` subroutine is complete, the entire application should be ready to run.

Step 5:
Empirically measure the accuracy of your application over an extended period, and answer the following "Thought Questions":

(a) How much error accrues (vs. observed "wall clock" time) after 10 minutes of operation? After 30 minutes of operation? Is your application running "fast" or "slow"?

_____

_____

_____

(b) Based on the error measured in (a), estimate how far off your application will be after 24 hours of operation? After one year of operation?

_____

_____

(c) List some potential sources of timing error and describe how you could "tweak" your application to make it more accurate. Also elaborate on the expected limits to the accuracy of your application.

_____

_____

_____

_____

