EE247 Project #2 Feedback, Comments, Corrections, Suggestions.

Dear class,

On these reports, I saw stuff all the way from really problematic to pretty darned good. If you are at the problematic end of the spectrum, it's important for you to include enough material for me to assess and perhaps understand what is going on. In some cases stuff didn't work but the report was "good" in the sense of communicating enough that I could understand the problems. In other cases there wasn't enough for me to look at; no telling what was happening. However, perhaps in those cases there were problems similar to what I saw elsewhere. So, in this document I intend to cover a variety of topics for which there seems to be widespread misunderstandings or problems.

1. Naming conventions.
   If someone else is going to look at your code, or you hope to understand it yourself later, you need to use variable, function, file and other names that are meaningful. It is hard to read code. Some people have a hard time with English; code is harder. Give them some help! Try to use names that are meaningful.

   Here's an example: For the terminal (the files hcc_terminal.c and hcc_terminal.h and associated code in your main file) there is a convention that commands, the command text, and the command callback function follow a naming convention. If the command text (that you type in to give the command) is "led", then the command (the particular command_t structure) is named "led_cmd" and the callback function is named "cmd_led". Why not follow that convention for your other commands?

   Another convention for variable names is to use lower case for local variables and a name with an upper case letter or two (but not the first) for globals. Or, in addition, start all global names with a "g". I've seen the convention that function names start with a Capital letter, and other things generally don't. Or even that function names associated with a particular "module" start with all caps for that module, like "TERMIO" functions that have to do with terminal input and output using the serial port in the files SCI_functions.c and SCI_functions.h. (The ICS functions in those files are technically for the clock module, only indirectly for the serial port.)

   It is OK to use "i" for an index or counting variable, or "n" for how far you count, and things like that which are common mathematical practice. But variables that mean something, like led state, should have names like "led_state", not just "x" or "y". The variable name should help the reader understand what the variable means. Similarly, the function name should communicate what it does. Don't use "ClockInit" for the name of a function that prints out the time. Or "F".

2. Comments.
   Help the reader by using comments. You don't have to comment everything. For example, if you have a statement: i++;, you're wasting your time to add the comment "/* adds one to i */". The reader can see that. But, what does "RTCSC=0x37;" mean? It is not obvious. Help the reader. Explain. Since the register RTCSC is made up of various fields, it is better to put it in binary, with explanation:

   ```
   RTCSC = 0b01100111; /* RTCLK=01, RTIE=1, RTCPS=7; 65.5ms interrupts */
   ```

Comments should also explain what a group of statements are doing where that isn't necessarily obvious. For example, in initializing a TPM time you may need several statements that you could explain with a single line comment:

/* Set TPM1C1 to be at 10KHz with a 30% duty cycle. No int. */

It is good practice to put comments before each function to explain what that function does. If there are arguments, explanations of them are helpful. Look at the headers for the functions in hcc_terminal.c. These are an example of what is expected if you are earning money and progamming is part of your reason for being paid. There are even more strict conventions for function and file headers in some systems. Normally every file should have a header that describes the module it implements, who is the author, the date, revision history and perhaps more.

Now, we are in an education setting. What you are looking at is not as complex as most real-world systems. But now is a good time to start doing a better job of documentation in your code. No, it is not a major point of emphasis in this course, because there are competing interests and we can't waste too much time doing documentation on things that are not so complicated that they need it on industrial scale. But, do start thinking about it, and help me out with comments while I am trying to help you with your code. It's like using wire color coding in Electronics. The electrons don't care. Your computer doesn't read the comments. But they are important to human communication.

3. Multiple files

I was hoping that with just the clock we could start practicing putting different "modules" in different files. I asked for a "clock.h" file and a "clock.c" file. The idea is to put the clock related code in "clock.c". That would typically be the clock initialization (where RTCSC is set), the functions to set and read the clock (the command callback functions). And, the Interrupt Service Routine (ISR) for the RTC timer. You would logically also put the time global variables in "clock.c". The command structures themselves present some interesting issues, because they are variable (constant variables).

The problem with putting all that stuff in "clock.c" is that you may need to call those functions or refer to those variables. That's why we have "clock.h". "clock.h" is the file where all the things in clock.c, at least those you need to reference from elsewhere, are declared. Declaration is a statement of something that exists, and what it looks like. Definition is where the name of the variable or function gets its meaning – the code or the contents.

Here's what might be in a typical "clock.h" file:

```
/*  Header – typically many lines including file name, author, date,*
*   etc.  Here just two lines in the interest of brevity.          */
#include “hcc_types.h”    /* and other includes that you may need */
# include “hcc_terminal.h” /* needed if you want to call print() */
void ClockInit(void);  /* call to start clock and RTC ISR */
static void cmd_time(char *param);/*command callback for reading time*/
static void cmd_set(char *param); /*command callback for setting time*/
```

The corresponding "clock.c" file would then have:

```
/*  Header – typically many lines including file name, author, date,*
*   etc.  Here just two lines in the interest of brevity.           */
#include “derivative.h” /* Needed to get register definitions */
#include “clock.h”    /* includes hcc_terminal.h as well */

/* Globals */
char gHours, gminutes, gSeconds;  /* basic variables for time */
int gMsec;                        /* millisecond counter */

void ClockInit(void){  /* call to start clock and RTC ISR */
  ….(all the code, including setting globals and RTCSC)….
}

static void cmd_time(char *param){/*command callback for reading time*/
 (the cmd_time code)
}

static void cmd_set(char *param){ /*command callback for setting time*/
 (the cmd_set code)
}

Interrupt 25 void rtc_isr(void){
      RTCSC_RTIF = 1;  /* reset flag */
      (and the rest of the code to advance the time)
}
```

What gets messy are variables. Suppose we wanted to get to gHours, gMinutes, and gSeconds from main.c. We need to declare them in main (or in a ,h file that main includes). But they can't duplicate definitions, such as those in "clock.c" above, because that would look to the compiler (and linker) like two globals with the same name. What we'd have to include in main.c is:

```
extern char gHours, gMinutes, gSeconds;
```

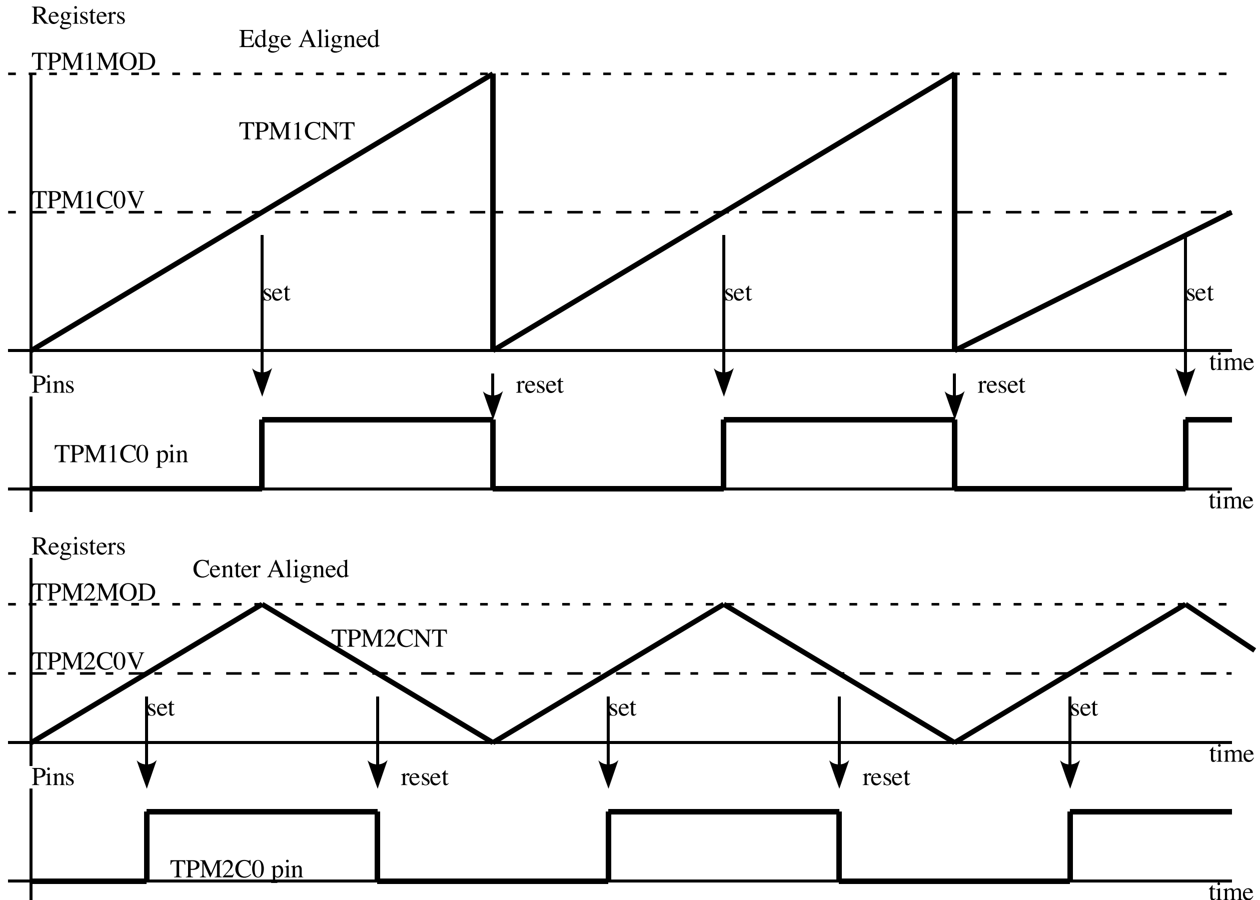
This declares that there are global variables with these names in some OTHER .c file. We can't put this in clock.h if clock.c is going to include clock.h, because ehten they'd be both defined and declared as externals in the same file.

The same applies to the commands for the terminal, because they too are global variables, though of type command_t rather than int or float or char. You could put the commands into clock.c, but then you'd need to declare them in main.c in order to reference them when you call "add_command":

```
Extern command_t time_cmd, set_cmd; /* commands to add */
```

Or, you could put the add_command calls into the ClockInit function in clock.c. (You'd have to make sure Terminal_Init was called before ClockInit though.) So, as you think about how to divide up a project into files for modules, look at hcc_terminal.

4. Using the TPM timer to get overlapped square waves:
   I saw two basic approaches to doing the TPM timers to get the overlapped square waves. Both work. The first method sets up one TPM as edge aligned and the other as center aligned, both with the same period and the same 50% duty cycle:

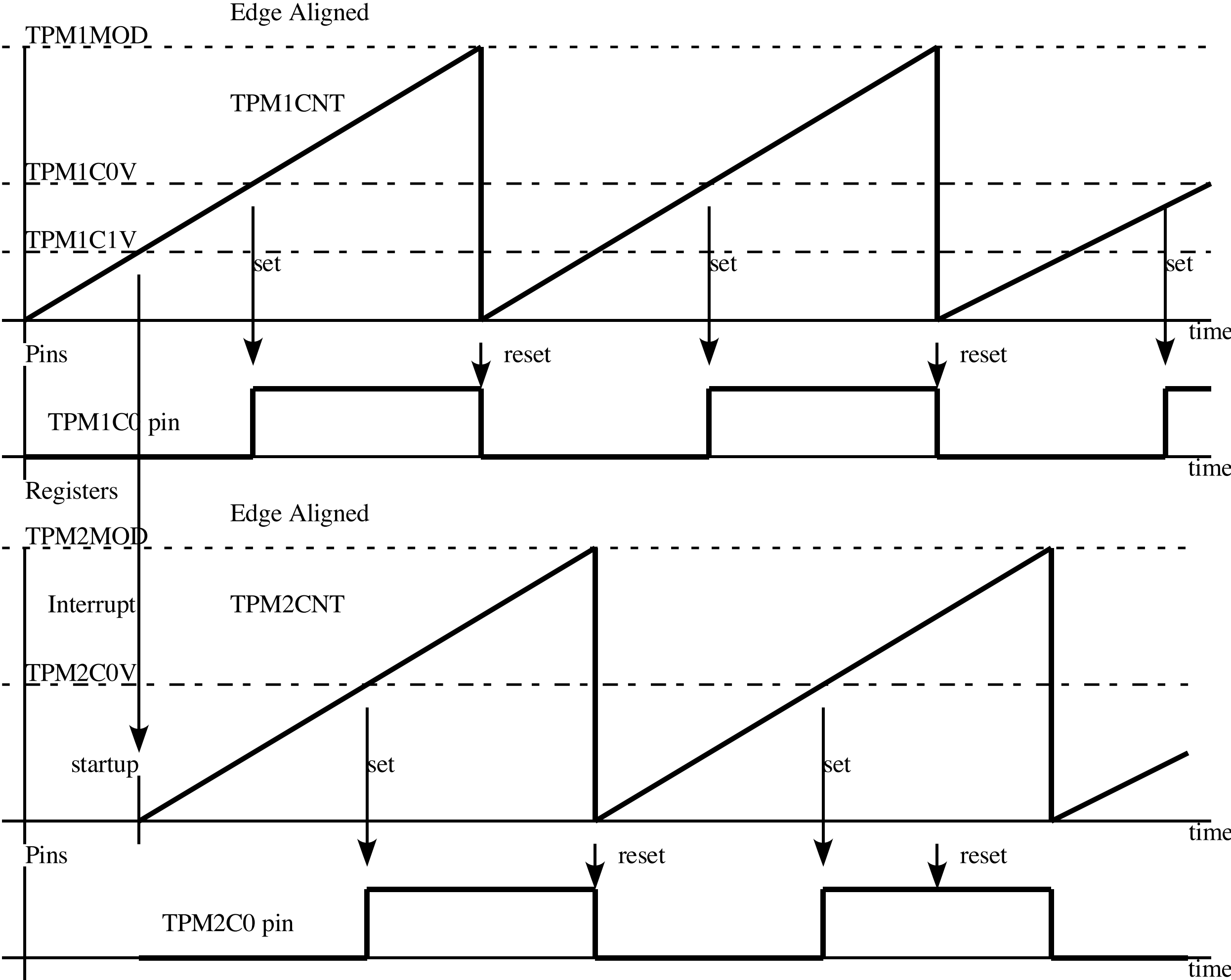


To make this work, the edge aligned TPM1MOD of the first channel must be an odd number (the number of clocks per cycle is TPMMOD+1) so that the period of the second channel can be exactly half of it. So, TPM2MOD = (TPM1MOD+1)/2. Remember, the TMP1CNT counter will count up to the TPMMOD value, starting with 0, so the number of counts per cycle is equal to TPMMOD+1. But center aligned counts up and down. So, it hits 0, and it hits TPM2MOD, only once each in each cycle. So the period of the waveform is (TPM2MOD-1)*2. That makes the period of the waveforms come out exactly the same. If they are not exactly the same the phase relationship will drift. (Even so, it might be a good idea to have a background task to keep an eye on these to check and see that the counters stay aligned.) To get exactly a 50% duty cycle for TPM1 (edge aligned) you want its channel value to be half of the TPM1MOD+1 value (the period in counts). Similarly, for TPM2. Using a large value or the MOD registers helps if the waveform is not exactly 50% due to rounding. But the periods do need to match exactly.

The advantage of this method is that the start of both cycles takes place at the same time. You can set up both TPM's and turn them on at the same time. If you want to adjust the frequency, you can set in interrupt on either and have the update to the MOD and channel values

take place at the same time for both. (Reversing is easier; just flip the assertion of one waveform or the other.) Credit to Matt Costello for working out and using this method!

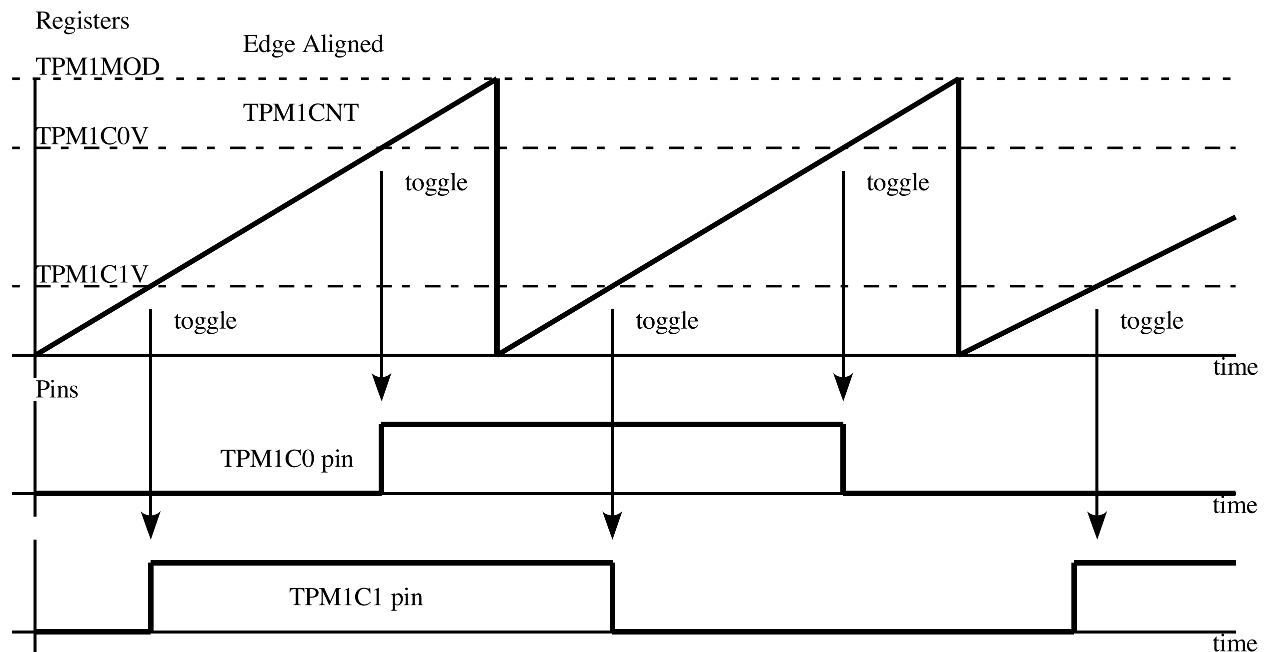
A more straightforward method is to use two edge aligned TPM's with identical settings, but start the second one 25% of a full period after the first one. The figure below illustrates.



The trick in doing this is how to delay the second TPM just the right amount. Here's my suggestion: Use a second channel of the first TPM with the channel value of 25%. The pin isn't used, just the channel value, but the channel is set to generate an interrupt when TPM1 is turned on. (The TPM2 MOD and Channel value can already be set, ready to go. When the interrupt fires, the TPM Channel interrupt service routine immediately starts up TPM2 (by setting the TPM2SC). It also turns off the flag, turns off the interrupt, and then won't fire again. The waveforms are off and running. If the PWM frequency is high, the interrupt channel value could be a little lower to compensate for the interrupt service time getting the second TPM started. TPM is a relatively high priority interrupt, so, especially during startup, there wouldn't be much delay.

In both cases, when speed is changed (by scaling the MOD values down or up, and possibly by changing prescaler values as well), the channel values need to be changed correspondingly. That should be synchronized so that both channels are updated at a "safe" time. A second channel could be used to generate an interrupt at a time when both channels are stable, or the TPM interrupt could be used. Both TPM's would be updated at the same time with values already on hand (perhaps stored in globals by a command function) with a global flag used to indicate that an update is needed. Or, the command function could, before it exits, enable the TPM overflow flag (for the end of the cycle) to notify the need for the update by enabling the ISR that does the update. This kind of signaling is needed to communicate among "processes" that run asynchronously on your computer. Yes, "multitasking". That is what we are going to be doing. More about that later.

There is a third possibility for doing TPM overlapped square waves using just one TPM with two channels, but it is tricky. The diagram below illustrates:



This scheme has the advantage of using just one TPM, and so it is simpler to get started and to adjust. The channel values are 25% and 75% of the period count (TPMMOD+1) minus 1. The one problem is that when adjusting the waveforms, you'd like to be sure to maintain the same phase relationships of the outputs. If you were to miss a toggle on one or the other, the direction will flip. Not good. Having updates take place on the TPM overflow interrupt is probably the safest way to do it. (Notice that with this method the waveform period is twice what it is for the other methods.)

In conclusion, those are the main points I'd like to make coming out of project 2. I'd like to encourage you to now go back and "fix" your project 2. Get the satisfaction of seeing everything work satisfactorily. That will help as we transition to Project 3, where just about everything is going to be working off of interrupts.