Guidelines for lab assistants

Below you find some useful information for your work in the laboratory, based on the course examination criteria and based on experience from the labs during earlier years.

Regarding the grading of the lab assignment

Unlike in most other courses the grading of this lab assignment follows a non-binary scale (i.e. U, 3, 4, 5). To facilitate the grading the students will be evaluated during the lab session with respect to four different performance aspects:

Implementation: How many of the coding challenges in Part 2 that the lab group can successfully implement and demonstrate. To be awarded the minimum grade of 3 the group must be able to *fully implement Part 2, Step 3*.

Design: How well the lab group knows the design and behavior of their own software code. To be awarded the minimum grade of 3 the group must be able to, *for the most part, adequately explain* the purpose of variables, objects, methods, etc.

Debugging: How well the lab group can identify, and solve, problems with the software code. To be awarded the minimum grade of 3 the group must be able to *adequately formulate symptoms* for problems with the code. To be awarded a higher grade the group must also show that they are able to find remedies to the problems.

Paradigm: How well the lab group understands, and can make use of, the reactive, concurrent, object-oriented and timing-aware programming paradigm used by Tiny-Timber. To be awarded the minimum grade of 3 the group must be able to *adequately understand and make use of the paradigm.*

In the normal case the members of a lab group will receive the same lab performance grade. However, in case of significant imbalance (with respect to programming skills, dedication to the assignment, etc) it is possible to give individual grades.

Common view of how to keep the lab group members active

In order to use your time wisely, and to facilitate the grading of the lab group, please use the following guidelines.

- Minimize the time spent with lab groups that are not able to formulate symptoms for possible problems with their software code. Kindly tell them to come back and ask for help when they have a clearer problem description.
- Minimize the time spent with lab groups that have run into problems with their software code because they did not read necessary instructions in the lab PM. If they are uncertain of how to interpret the instructions help them out. Otherwise, kindly tell them to thoroughly read the instructions in the lab PM and make a new attempt writing the code.

- Minimize the time spent with lab groups that have run into problems with their software code because they did not follow the software design guidelines advocated by the course material. Kindly tell them that they need to reconsider their design strategy, and refer them to the course material and the grading criteria.
- If you detect significant imbalance in a lab group, e.g. only one of the students seems to do the talking and the software coding, make an attempt to activate the other student by requesting that student to answer your next question, to describe the functionality of a certain piece of code, to give a demonstration of how the software is used etc.

Note: if any of the issues listed above keep re-occurring for a particular lab group make a note and let Jan know as soon as possible as this may affect the performance grade of the lab group.

Common view of certain software design choices

In order to guide the lab groups in a consistent way in their software design please use the following recommendations (most of which are also stated in the course material).

- The tone-generator task and the background task must use two separate objects in order to guarantee concurrency, which in turn is necessary to clearly illustrate tone distortion as well as the usefulness of deadline-based scheduling.
- Variables and data structures that are used by concurrent tasks should be placed inside an object, and be accessed only via mutex method calls (SYNC for reading, SYNC or ASYNC for writing).
- Big data structures, such as lists and tables, with non-changing contents may be defined as global variables and referred to directly (without involving an object).
- A gap between playing tones can be implemented either by muting the audio of the tone-generator task or by killing (and later restarting) the tone-generator task.
- The use of dynamic memory allocation may be used, but should not be encouraged unless the group is clearly knowledgeable of how to write code for it and is aware of its potential drawbacks in a time-critical application.

Common view of how to approve the solutions

In order to approve the solutions of the lab groups in a consistent way during the lab sessions please use the following answers and guidelines.

Part 0:

- Problem 3.a: 2500 μs
- Problem 3.b: 500 μs
- Problem 3.c: 1136 μs
- Problem 3.d: $p(k)$ = 440 × $2^{\frac{k}{12}}$
- Problem 3.e: a) 1702 μs b) 675 μs
- Problem 3.g: Maximum is 9 Minimum is -5
- Problem 3.h: `-5 -3 -1 -5 -5 -3 -1 -5 -1 0`
- Problem 3.i: Maximum is 14 Minimum is -10
- Problem 3.j: Here, frequency values are rounded, and period values truncated:

```
Index   Frequency f [Hz]       Period 1/(2f) [us]
--------------------------------------------------
-10             247                    2024
-9              262                    1911
-8              277                    1803
-7              294                    1702
-6              311                    1607
-5              330                    1516
-4              349                    1431
-3              370                    1351
-2              392                    1275
-1              415                    1203
 0              440                    1136
 1              466                    1072
 2              494                    1012
 3              523                     955
 4              554                     901
 5              587                     851
 6              622                     803
 7              659                     758
 8              698                     715
 9              740                     675
10              784                     637
11              831                     601
12              880                     568
13              932                     536
14              988                     506
```

- Make sure that, for problem 2.b, the state variables have been moved into the `App` object, and that `self` is used to refer to them.
- Make sure that the period values in problem 3.j are stored as constants in a table and used by the software as such. The values are preferably derived off-line (e.g. via Excel or MATLAB), but could also be derived at run-time and stored in the table (e.g. using floating-point calculations in the `StartApp()` method).
- When the students are finished with Part 0, remind them to not dispose of the program code. Most of it will be useful in the later parts.

Part 1, Step 1:

- Remind the students to be cautious when the software volume control is tested for the first time (e.g. use the oscilloscope.) **NO earphones on/in the ears!!!**
- Encourage the lab group to use a separate object for the tone generator. Inform them that the tone generator state variables (e.g. period, volume, mute) must in any case be stored within an object, and not as global variables.

  If a separate object will be used for the tone generator check with the lab group whether they know that suitable methods must be defined for accessing/modifying the tone generator state variables, and that these methods should be called via SYNC/ASYNC. Discuss with the lab group what is the most suitable object to store the state variables (hint: the frequency of updates/accesses speaks in favor of storing inside the tone generator object).
- Measure the frequency of the tone generated in the speaker, and verify that it is 1 kHz. Use the oscilloscope, or a guitar tuner app or a frequency analyzer app on your phone. If unfamiliar with these tools, let someone else do the measurement.
- Discuss with the lab group how they chose to implement the mute function.

  Since the tone generator must be running continuously, the approaches to use are: (i) to not update the DAC at all while muting, or (ii) to write the value 0 to the DAC while muting. If alternative (ii) is used, it is important that the previously set volume level is restored when un-muting.

Part 1, Step 2:

- Check the lab group's implementation of the background task. Make sure that the background task uses a separate object, that the corresponding state variables (e.g. period, load) are stored within the object, and that SYNC/ASYNC calls to suitable methods are used when the state variables are accessed/modified from outside of the object.
- Make sure that the distortion generated can be clearly heard when the load is increased, and that the character of the distortion is different in problems 2.a, 2.b and 2.c. If not, check whether the two tasks are really using two separate objects and that the period of the background task is correct (1300 μs).
- Discuss with the lab group why the distortion occurs.

  Answer: The background task disturbs the execution of the tone-generator task, since there is no special priority given to the latter task.
- Check the distortion with an oscilloscope for additional visual insight. If you cannot handle an oscilloscope, let someone else (e.g. Jan) do it.
- Discuss with the lab group why the distortion in problem 2.b is so much different from the distortion in problems 2.a and 2.c.

  Hint: Let them consider the period of the tone generator in problem 2.b.

Answer: In problem 2.b the periods of the tone generator (650 μs) and the background task (1300 μs) are harmonically related, and therefore produce a stable waveform (which can be seen on the oscilloscope). As the background load is increased the generated waveform remains stable, but becomes increasingly asymmetrical until just a very narrow pulse remains. This can be heard as the original tone (650 μs) changing its character to a more and more high-pitched sound, until it finally can no longer be heard by the human ear. The periods in problems 2.a and 2.c are unrelated to the background task, which means that the generated waveform will always be heard, but becomes more and more unstable as the background load increases.

Part 1, Step 3:

- Check the lab group's implementation of the deadline enable/disable function. Make sure that the corresponding state variable is stored within an object, and that a SYNC/ASYNC call to a suitable method is used when the state variable is accessed/modified from outside of the object.

- Make sure that no distortion can be heard when the value of variable `background_loop_range` is in the range $[1000, 8000]$.

- Discuss with the lab group why there is no distortion in this range.

  Answer: The tone-generator task is now always given a high priority (urgent deadline) when it executes and will therefore not be disturbed by the background task who has a lower priority (less urgent deadline).

- Make sure that the pitch drop effect can be clearly observed when the value of variable `background_loop_range` is in the range $[10000, 21000]$.

  The exact loop range value at which the effect kicks in depends on the C code that implements the loop in the background task. A sloppily designed `for` loop yields the lower value whereas a cleverly designed `while` loop yields the higher value. The most commonly-reported loop range value is 13500 (a standard `for` loop).

  Note: Make sure that the lab group does not confuse the pitch drop effect with the distortion that appears just before the actual pitch drop.

- Discuss with the lab group why the pitch drop effect occurs.

  Answer: As the execution time of the background task exceeds its period (overrun) the task retains its (absolute) deadline. Any new instance of the tone generator that occurs while the background task continues its overrun will have a deadline further away and therefore lower priority. Once the background task completes its overrun, the tone generator will execute although delayed. Then the background task will execute again, and once again overrun causing yet another delay of the tone generator execution. This repeated overrun execution causes the tone generator to get a real period that is longer than intended, resulting in a lower frequency of the generated tone.

Part 1, Step 4:

- Make sure that the measured WCET for the background task is in the range $[1250, 1350]$ μs when the value of variable `background_loop_range` from problem 3.c is used (i.e. when the pitch drop effect kicks in).

  Discuss with the lab group why it is reasonable that the WCET of the background task has a value in the range $[1250, 1350]$ μs in this case.

  Answer: Because the WCET of the background task is then very close to the period of the task (1300 μs), thereby consuming almost all processing capacity and significantly disturbing the tone generator task.

- Make sure that the measured WCET for the background task is in the range $[80, 120]$ μs when the value of variable `background_loop_range` is 1000.

  Discuss with the lab group why it is reasonable that the WCET of the background task has a value in the range $[80, 120]$ μs in this case.

  Answer: The fraction between the loop ranges for the above WCET values should be the same as the fraction between the WCET values themselves. Example: If 13500 loops cause the pitch drop effect in problem 4.a and 1000 loops is used in problem 4.b, then the WCET in problem 4.b should be approximately 13500/1000 = 13.5 times lower than the WCET in problem 4.a.

- Make sure that the measured WCET for the tone generator is less than 1 μs <u>but non-zero</u>. Typically, the WCET should be in the range $[100, 300]$ ns.

  Discuss with the lab group what is a reasonable value of the tone generator WCET.

  Answer: Given that the code is very short (only involving writing a new value to the DAC) and that the processor frequency is 168 MHz, they should come to the conclusion that the WCET must be less than $1\mu s$.

  Note: If the measured WCET is exactly 0, the reason is most likely that the lab group has not reflected upon the resolution of the system clock ($10\mu s$). In order to measure the WCET accurately a special measurement solution is required, e.g. changing the time scale by running the measured code 1000 times between the begin and end samples (giving values in ns units). Let the lab group think about this for a while before you give them a hint.

- Comments regarding measurements made with a system clock resolution of $10\mu s$:

  - If reported WCET values are consistently 10 times lower than the expected values a time conversion macro is probably missing (e.g. `USEC_OF`).
  - If reported WCET values are consistently 100 times lower an erroneous time conversion macro is probably used (e.g. `USEC` instead of `USEC_OF`).

- When the students are finished with Part 1, inform them that they no longer need the code for the background task and the WCET measurements. They should make a backup of the files and then remove that part of the code.

Part 2, Step 1:

- Make sure that the lab group's software design sketch is illustrated using access graphs and timing diagrams.

  The timing of the the gap event and the next-played-note event must be clearly shown, and be put in relation to the baseline of the currently-played note.

  The timing diagram should illustrate the application's full concurrency. This means that method executions may be overlapping in the diagram, while scheduling-related details such as local skew or preemptions should not be shown.

- Make sure that independent tasks in the design are represented by separate objects.

  The lab group should realize that they need a separate object for handling the playing of the melody, to implement a state machine that traverses the 32 elements of the frequency index and note length arrays in a repeated fashion. The tone generator task should not be involved in anything else except generating a tone!

- Discuss with the lab group their choice of objects for storing the tone generator parameters (period, deadline, volume) and melody player parameters (tempo, key).

  Check that the lab group is aware of the correct approach regarding how these parameters should be accessed and updated from outside the object. Answer: via SYNC/ASYNC/AFTER calls to suitable methods.

- Discuss with the lab group their approach for implementing the gap after each played note. Is the tone generator killed and restarted, or is it muted?

  Check that the lab group has a solution where it is the melody player that initiates the gap event (similar to the example in Exercise session 3.) The tone generator should not poll the system clock to know it is time for the gap.

Part 2, Step 2:

- Measure the default tempo of the melody, and verify that it is 120 bpm. Use a metronome app on your phone, or let someone else (e.g. Jan) with such an app do the measurement. Also check other tempos, such as the extreme values (60 and 240 bpm) as well as some random value within the domain (e.g. 131 or 209 bpm).

  Note: Make sure that the tempo parameter is read from the keyboard as an integer number (bpm), and not as an increment/decrement command. The lab group software should be able to translate between beats-per-minute (bpm) and note length (ms).

- Measure the default frequency of the first melody note, and verify that it is 440 Hz. Use a guitar tuner app or a frequency analyzer app on your phone, or let someone else (e.g. Jan) with such an app do the measurement. Also check that the melody can be played in other keys (e.g. −5 and +5).

  Note: To be able to take a good measurement of the first note the melody should be played at the slowest possible tempo (60 bpm, or lower if possible).

Note: Due to the resolution of the real-time clock the frequency of the generated tone will not be exactly 440 Hz. The measured frequency will be closer to 442 Hz.

Note: Make sure that the key parameter is read from the keyboard as an integer number, and not as an increment/decrement command.

- Make sure that the melody is played in a cyclic fashion, and that the tempo is maintained also in the transition between the last note of one melody repeat and the first note in the subsequent melody repeat.

- Make sure that there is a short gap between the notes. This is easiest to observe between two subsequent notes with the same frequency, e.g. between note 4 and 5 in the melody, or between the last note in one melody repeat and the first note in the subsequent melody repeat.

- When the students are finished with Part 2, Step 2, remind them that they should submit their software code in Canvas. Normally, they only need to submit the `application.c` file, but if they have made changes to multiple files they should put them into an archive file (.zip or .tar) and submit that file.

Part 2, Step 3:

- General comment: If the lab group has not yet settled for a common CAN protocol together with other groups, a simpler protocol is acceptable to use for this step.

- Make sure that the contents of each received CAN messages are printed out in both leader and slave mode. But only when a cable is connected!

- Make sure that the key and tempo parameters are sent as integer numbers in the protocol, and not as increment/decrement commands.

  Note: Check particularly that negative values of the key parameter (e.g. -5) can be handled by the protocol.

- Make sure that the melody can be controlled (start, stop, change tempo and key) from the keyboard in both leader and slave mode when the CAN cable is connected. When the cable is disconnected it should only be possible to control the melody in leader mode.

  Hint: In slave mode, disconnect the CAN cable and let the group give 2–3 commands from the keyboard. With TinyTimber's default configuration the board's CAN controller chip will buffer up to three CAN messages. When the cable is reconnected the buffered messages will be transmitted and the keyboard commands be executed as a batch.

- Make sure that the slave mode works in an *semi-autonomous* fashion; that is, once the playing of the melody has been initiated from the keyboard the playing should continue even if the CAN cable is later disconnected. See demonstration procedure in Problem 3.b in the lab PM.

Part 2, Step 4:

- General comment: It is required that three boards should be able to play the melody in either form (chorus or canon). If there are less than three lab groups collaborating, one of the groups should use their software on multiple boards.

  Note: It is not required that the boards shall be able to switch dynamically between leader and slave mode. Separate software versions may be used for each mode.

- Make sure that the melody is played in a cyclic fashion in *chorus form*, for different initial tempos and keys. Also make sure that the note synchronization is maintained between the boards after several repeats of the melody. The melodies of the boards should not drift apart!

  Note: Due to delays in the CAN communication there may be a slight (barely noticeable) delay between notes that are supposed to be played simultaneously by the boards. If there is a noticeable "echo" delay between the boards there is a problem with the software in one or more of the slave boards. A probable cause is that the software prints out debug text for each received CAN message before it starts generating the notes.

- Make sure that the melody is played in a cyclic fashion in *canon form*, for different initial tempos and keys. Also make sure that the note synchronization and the "canon separation" (parameter n) is maintained between the boards after several repeats of the melody. There should be no board playing in unison with another board.

  Note: It is not required that the boards shall be able to switch dynamically between chorus and canon form. Separate software versions may be used for each form.

  Note: It is only required that the boards shall be able to play the melody in canon form with parameters $n = 4$ and $n = 8$. Making things work with other values of n is trickier than one might think.

- Make sure that a change of key or tempo must be adopted by all connected boards within a *recovery time* corresponding to the length of 12 beats, measured from the time the leader requests the corresponding change.

- Make sure that the note synchronization and the "canon separation" (parameter n) is maintained in canon form between the boards for different *dynamic changes* in tempo and key.

- The Ultimate Test: check that the nodes do not lose synchronization in canon mode after the following tempo-change scenario:

  1. start with a tempo of 120 bpm
  2. wait for a little while
  3. increase the tempo to 240 bpm
  4. wait for a little while
  5. decrease the tempo to 60 bpm

Other good-to-know type of information

MD407 debug monitor and TinyTimber specifics:

- The monitor command 'go' is a shorthand version of the command 'go 20000000'.
- As of version v2.05 the TinyTimber kernel has enabled support for the detection of divide-by-zero and unaligned data transfer hardware faults. If any of these faults occur the program will abort and return to the MD407 debug monitor.

  An unaligned data transfer means that a 16- or 32-bit word is read from or written to an odd memory address. This could happen e.g. if there is an attempt to write a 32-bit data word into the CANMsg data structure's 8-byte buffer (which begins at an odd byte offset within the structure).
- If the message "PANIC!!! Empty queue" appears on the console output the TinyTimber kernel is temporarily overloaded by interrupt requests. Probable causes: a key on the workstation keyboard has been pressed for too long (activating auto repeat), or some computer board has transmitted CAN messages too frequently.
- If the message "PANIC!!! Empty pool" appears on the console output the TinyTimber kernel cannot keep up with scheduling requests. Probable causes: a method is repeatedly calling itself with AFTER using an offset of zero, or multiple copies of the tone generator task exist because old copies were not successfully terminated.
- If a program is terminated (due to a crash, or via the RESET button) and returns to the MD407 debug monitor the program code should be downloaded again in order to guarantee the consistency of initialized static variables (.data segment).

CodeLite specifics:

- It is good practice to always aim at having zero warnings after compilation. A warning may refer to type mismatches which, in this type of programming, may cause serious problems.
- Printing of floating point numbers with `sprintf %f` is disabled by default (it outputs nothing) in order to keep code size down. It is recommended that floating point numbers are type-cast to integers and then printed with `sprintf %d`. Printing with `sprintf %f` can be enabled by adding the compiler flag '`-u _printf_float`'.
- If the lab group has problems building their project under Windows, a common cause is that the directory path for the project files contains a space (' ') character.
- If the lab group has problems with the CodeLite configuration (e.g. strange compiler setup), it could mean that they have used CodeLite in some other project. Restore CodeLite to its default configuration by erasing the (hidden) directory

  ```
  C:\Users\cid\AppData\Roaming\codelite
  ```

  On some machines the (hidden) directory is instead

  ```
  Z:\.win\AppData\Roaming\codelite
  ```

  After removing the directory restart CodeLite and let it do a default setup.

CoolTerm specifics:

- It is possible to use drag-and-drop of a file from an OS file browser window (e.g. Explorer, Finder) to a CoolTerm window. This saves a lot of clicks/keypresses in the long run.
- Strange things: As of CoolTerm version 1.4.6 for Windows the '.s19' load files are no longer displayed using the 'All files' option in the CoolTerm file browser.

  Solution: Use the version (1.4.5) of CoolTerm that is available on the course page, or use the drag-and-drop functionality (see above).

Particularly nasty bugs:

- Watch out for the combination of `sprintf()` and character arrays. It happens frequently that data is written beyond the limits of a too modestly dimensioned character array, which in turn may cause many hard-to-solve problems in the software.
- Watch out for the use of `&self` instead of `self` in TinyTimber operations, such as ASYNC, AFTER, etc. This is a serious case of type mismatch and will most likely cause a memory fault. Surprisingly enough, the compiler does not give a warning or error for this type mismatch.
- Watch out for calls to ASYNC, AFTER, etc that crashes the program without any apparent cause. It may happen if the object referred to in the call is missing an 'Object super' as the first element in its C struct.
- Watch out for local data structure variables within a method that are being used for storing CAN messages. The contents of the data structure becomes volatile (most likely overwritten) as soon as the method call completes.
- Watch out for CAN messages transmitted with unknown contents of the length field. A very hard-to-find bug in the receiving software.

Recommended tuner and metronome apps:

- TonalEnergy Chromatic Tuner and Metronome (40 SEK in iOS App Store)
- Cleartune (40 SEK in iOS App Store)
- Metronome: Tempo Lite (Free in iOS App Store)
- ...