EDA223/DIT162 Real-Time Systems January 6, 2025
Laboratory Assignment — Part 2

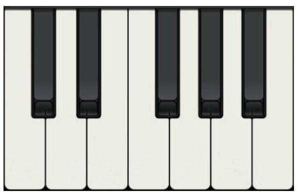
Objective: The purpose of Part 2 of the lab is to design a music player capable of performing the song Brother John. First, you will develop the simple tone generator of Part 1 into a small synthesizer/sequencer combo that can play the melody on your board (Step 1 and Step 2). You will then further develop your player to allow it to be controlled via the CAN bus (Step 3) as well as via your board's USER button (Step 4).

Approval: When you see the text "*Assistant's approval: ....*" below a problem description you should show your solutions to the laboratory assistant. If the solutions are found to be satisfactory the laboratory assistant will mark the corresponding examination objective as 'Passed' in Canvas. You can then continue with the next problem.

Step 1: The Brother John player – the design

Starting with the tone generator from Part 1, you should now extend your software design with the functionality of playing notes of varying frequency and length.

The frequency of a note is expressed in terms of a ***frequency index***, where a frequency index of 0 represents the ***base frequency***. Varying the frequency of a note means choosing among the tones in the Western 12-tone scale. Recall that, on a piano keyboard, there is a pattern of 12 keys (7 white and 5 black) that repeats itself over the entire keyboard. These 12 keys constitute an octave.



The relation between tones that are an octave apart on the keyboard is well defined: their frequency ratio is exactly 2. The frequencies of the remaining tones in the scale can be defined according one of the many temperament systems that are available. The temperament system used in this laboratory assignment is the equal-tempered 12-tone scale, which consists of a range of frequencies $p(i)$ (where i is the frequency index) such that the ratio $p(i+1)/p(i)$ always equals the twelfth root of 2. The default base frequency in this assignment is $p(0) = 440$ Hz.

The Brother John melody is a (cyclic) sequence of 32 notes, with the frequency of each note being defined by the following list of indices:

```
0 2 4 0 0 2 4 0 4 5 7 4 5 7 7 9 7 5 4 0 7 9 7 5 4 0 0 -5 0 0 -5 0
```

The duration of a note is expressed in terms of the length of a ***beat***, which in turn is determined by the melody's ***tempo*** as measured in beats per minute (bpm). All notes in the Brother John melody are not of equal length, but exhibit the following (cyclic) pattern:

$a \; a \; a \; a \; a \; a \; a \; a \; a \; a \; b \; a \; a \; b \; c \; c \; c \; c \; a \; a \; c \; c \; c \; c \; a \; a \; a \; a \; b \; a \; a \; b$

Here, a represents a length of one beat, b represents a length of two beats (i.e., $2a$) and c represents a length of half a beat (i.e., $a/2$). The length of a beat is set by selecting a tempo for the melody. The default tempo in this assignment is 120 bpm, which means that the default length of a beat is 500 ms.

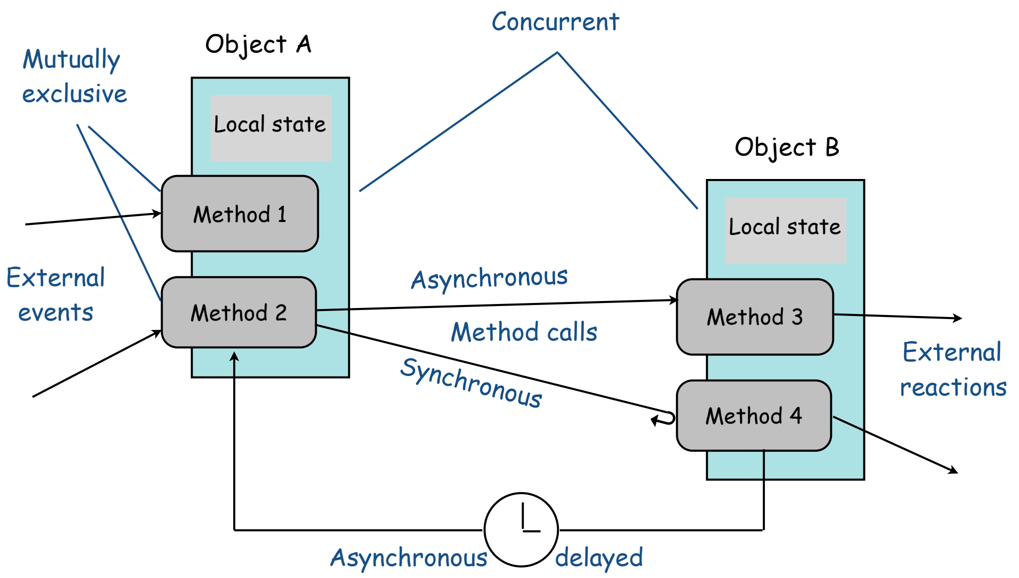


Figure 1: An example of an access graph

To give the melody some character a small gap of silence of suitable length should be inserted at the end of each note, by shortening the actual note duration by the same amount. A suitable length of the gap is 50–100 ms.

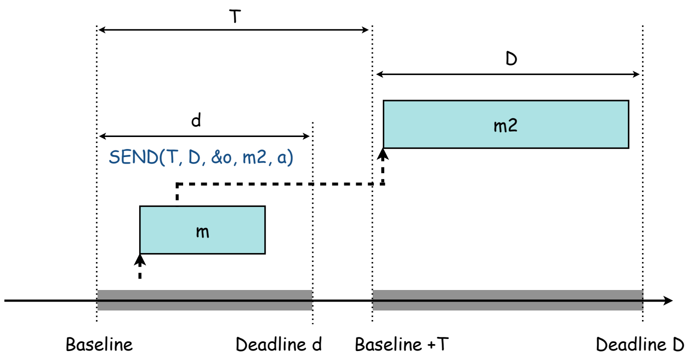


Figure 2: An example of a timing diagram

The performance of Brother John melody is controlled by the parameters `key` and `tempo`. The value of `key` is used as an offset to the frequency index of each note, thereby transposing the melody from the default key of A (parameter `key` = 0) to any other key within a given range. The value of `tempo` determines the length of a beat.

To be able to play the Brother John melody you should now create a software design that consists of (i) a music player that controls the behaviour of (ii) a tone generator. Since the music player and the tone generator have different roles (responsibilities) in playing the melody they should be designed to use two independent task objects.

IMPORTANT! Before you begin writing any code you should get your design ideas approved by the laboratory assistant. To illustrate your design you should use access graphs and timing diagrams, similar to the ones shown in Figure 1 and Figure 2.

Problem 1.a: Using timing diagrams illustrate how the tone generator will be controlled by the music player to (i) play a note of a certain length, including the small gap of silence, and (ii) play the next note in the melody after the gap.

Problem 1.b: Using access graphs illustrate (i) how the tone generator will be controlled to play a note of a certain length, including the small gap of silence, (ii) how the tone generator will be controlled to play the cyclic sequence of 32 notes constituting the Brother John melody, and (iii) how the key and tempo of the melody can be dynamically changed from the keyboard while the melody is being played.

Assistant's approval: ....

Step 2: The Brother John player – the code

You should now create a software implementation of the music player design that you developed in Step 1.

Define suitable data structures for storing the frequency and length information[1], and write code for the methods shown in your access graphs and timing diagrams that were approved in Step 1. Make sure that your `reader` function is able to dynamically supply the `key` and and `tempo` parameters to the relevant software parts.

The special requirements of the music player software implementation are as follows:

1. The music player should use a different object than the tone generator.
2. The music player should automatically start playing the melody in a cyclic fashion, using the default parameters `key` = 0 and `tempo` = 120 bpm.
3. The music player should be able to change key and tempo dynamically while the melody is being played.
4. The `key` and and `tempo` parameters should be read from the keyboard in the form of integer numbers[2].
5. The music player should able to handle a `key` parameter in the range of $[-5 \ldots 5]$, and a `tempo` parameter in the range of $[60 \ldots 240]$ bpm.
6. The music player should include volume up/down control and mute functions[3].

Problem 2: Demonstrate that your software implementation of the music player is able to play the Brother John melody, and that key, tempo and volume can be dynamically changed from the keyboard while the melody is being played in a cyclic fashion.

Assistant's approval: ....

IMPORTANT! After approval you should submit your software code in Canvas (under 'Modules'/'Laboratory Software'). The code should be submitted as is, that is, in the shape it was at the time of approval (without modifications).

Create a '.zip' archive file, containing all your modified C files, and name the archive file '`Part2_2_PGID.zip`' (where `PGID` is your project group name). Typically, you only need to include the `application.c` file in the archive file, but if you have created additional files you should include them too.

A short `User's Guide`, describing how to control the software with the key commands, should be included (by means of C program comments or console print-outs) in `application.c`.

[1]Reuse the array of pre-computed tone generator periods that you defined in Part 0.
[2]Reuse the software code for reading integer numbers that you wrote in Part 0.
[3]Reuse the software code for volume control and muting that you wrote in Part 1.

Step 3: Yes, we CAN! (NOTE: Only required for grade 4 and 5)

You will now enable your music-playing board to send broadcast messages over the CAN bus. When you connect your board to the CAN bus your program must be prepared to run in either ***conductor mode*** or in ***musician mode***. In conductor mode your board's music player will be controlled exclusively via your keyboard. That is, whenever you give a command via the keyboard the conductor board's music player should react on the command, while it at the same time sends out a corresponding command on the CAN bus to each connected musician board. In musician mode your player will be controlled exclusively via incoming messages from the CAN bus, sent out by a conductor board. Moreover, a musician board should run in a ***semi-autonomous*** fashion; that is, once the conductor board has initiated the playing of the melody a musician board should be able to play the melody even if the conductor board does not send any additional commands.

You should first familiarize yourself with the CAN device driver. Locate the `receiver` method in the `application.c` file. Whenever a message is received from the CAN bus, the `can_interrupt` method (an interrupt handler) located in the `canTinyTimber.c` file is invoked and the message is read from the MD407 board's CAN controller. The interrupt handler then invokes the `receiver` method, within which you can acquire and decode the contents of the received CAN message.

To acquire the contents of the received CAN message, you should use the operation `CAN_RECEIVE()`. Correspondingly, to send a new CAN message you should use the operation `CAN_SEND()`. Locate the use of `CAN_RECEIVE()` and `CAN_SEND()` in the `application.c` file, and notice that they use a pointer to a variable of data type `CANMsg` as the second parameter. The data type `CANMsg`, defined in the `canTinyTimber.h` file, is used for storing only the user-relevant parts of the CAN message, such as the message identifier and the (maximum 8-byte) message payload. In TinyTimber's CAN device driver the 11-bit identifier is stored in two parts in the `CANMsg` type: one part consisting of the four least significant bits of the identifier (referred to as `nodeid`), and one part containing the 7 most significant bits of the identifier (referred to as `msgid`). This partitioning of the message identifier allows you and the other groups to enumerate your MD407 boards (via `nodeid`) and the transmitted messages (via `msgid`).

When designing your software it is important to know that <u>the CAN controller does not receive what it sends out</u>. In order to test your software, you will therefore make use of a ***loopback feature*** to verify that you can play in both conductor mode and musician mode.

To enable the loopback feature of the CAN device driver for your MD407 board you should proceed as follows:

- Connect the primary CAN interface (CAN1) to the secondary CAN interface (CAN2) by means of the dedicated white (loopback) cable that came with your loan equipment.

- Configure the CAN device driver by removing the comments on line 3 in `canTinyTimber.c` (that is, enable the `#define __CAN_LOOPBACK` statement).

As part of adapting your code to handle CAN messages you should add a *play function* to your program that allows you to <u>start and stop the playing of the melody</u>. A start command should initiate the playing from the beginning of the melody. When you run your program the melody should not be playing by default. Control the start/stop function with suitable keys on the keyboard. In addition, you should prepare the `receiver` method to print out the contents of messages received on the CAN bus regardless of which mode you are running in.

NOTE: Each transmitted CAN message should be *self-contained*, that is, the message should contain all necessary information to control the music player on a receiving board. This means that it is not allowed to send one character at a time in a separate CAN message (e.g. just echoing what you type on the keyboard) in order to change key or tempo of the melody.

Problem 3.a: Run your program in conductor mode, and make sure that you are able to control the melody (i.e. start, stop, mute, change key, tempo and volume) with your keyboard, regardless of whether the loopback cable is connected or not. Observe the printouts of the contents of each received CAN message, and verify that the messages that you receive (when the cable is connected) contain the same information as the messages you send.

Problem 3.b: Run your program in musician mode. Since you will not be connected to any other board you should use a special solution, in which you let your own keyboard send out CAN messages (just like you did in conductor mode.) Connect the loopback cable and make sure that you can control the melody playback from your own keyboard (start, stop, mute, change key, tempo and volume), and also observe the printouts of the received message contents.

Demonstrate the semi-autonomous functionality of your program as follows:

1. With the loopback cable connected initiate the playing of the melody from the keyboard.
2. Disconnect the loopback cable and show that you no longer can control the playing of the melody from the keyboard. That is, it should not be possible to stop the playing, or change key or tempo, of the melody. However, the melody should continue to play even though the cable is disconnected.
3. Reconnect the loopback cable and show that you once again are able to control the playing of the melody from the keyboard.

Assistant's approval: ....

IMPORTANT! After approval you should submit your software code in Canvas (under 'Modules'/'Laboratory Software'). The code should be submitted as is, that is, in the shape it was at the time of approval (without modifications).

Create a '.zip' archive file, containing all your modified C files, and name the archive file '`Part2_3_PGID.zip`' (where `PGID` is your project group name). Typically, you only need to include the `application.c` file in the archive file, but if you have created additional files you should include them too.

A short `User's Guide`, describing how to control the software with the key commands, should be included (by means of C program comments or console print-outs) in `application.c`.

Step 4: Tap that Tempo! **(NOTE: Only required for grade 5)**

Your final challenge is to equip your music player with some special functions, by means of a red USER button and a green LED on the MD407 board. To enable this functionality you need to install a small TinyTimber add-on software package, containing the `SIO` device driver.

Button call-back:

Your first challenge is to activate the USER button on the MD407 board, and allow it to interact with your software.

Download the `SIO` device driver files, `sioTinyTimber.c` and `sioTinyTimber.h`, from Canvas[4], and add them to your TinyTimber project in CodeLite. Then make the necessary changes in the `application.c` file to enable the device driver and allow it to make a call-back to a method of your choice whenever you press the USER button.

Apart from the automatic calls to your button call-back method, the following USER button functionality is available in the `SIO` device driver:

- With the operation `SIO_READ()` you may sample the current state (mechanical position) of the USER button: a return value of 0 means that the button is currently being *pressed*, while a return value of 1 means that the button is currently *released*.
- With the operation `SIO_TRIG()` you may configure the `SIO` device driver as to what event of the USER button will trigger a call to your call-back method: a parameter value of 0 means that call-back should happen when the button makes the transition *from released to pressed*, while a parameter value of 1 means that call-back should happen when the button makes the transition *from pressed to released*.

  The default functionality of the `SIO` device driver is to trigger a call-back when the USER button makes the transition from released to pressed.

Your software should support two modes of USER button functionality:

- The PRESS-MOMENTARY mode, where the button is *pressed and then released immediately*. This is the default mode of operation.
- The PRESS-AND-HOLD mode, which is entered if the button is *pressed and then remains pressed* for at least one second. Your software leaves this mode once the button is released, after which the default mode of operation applies.

Special software quality considerations:

- Your USER button software should make excellent use of the CORT properties of the TinyTimber kernel, in particular its support for concurrency, reactiveness and timing awareness.
- In order to fully exploit the inherent concurrency of the music player player, your USER button software should not use any type of blocking code (e.g. busy-waiting loops).

[4]The files are available on the 'Resources' page in Canvas, under the 'Software' tab.

The special requirements of the USER button software are as follows:

1. In the PRESS-MOMENTARY mode the software should measure the inter-arrival time between every pair of subsequent valid activations[5] of the USER button and print out the inter-arrival time expressed in milliseconds.

2. If the button is pressed and remains pressed for less than one second, the software should remain in the PRESS-MOMENTARY mode and immediately be responsive for new valid activations of the button as soon as it is released.

3. When the software enters the PRESS-AND-HOLD mode it should make a print out of the fact.

4. When the software leaves the PRESS-AND-HOLD mode it should measure the length of the time interval during which the button remained[6] pressed and print out the length expressed in seconds.

NOTE: Whenever you press the USER button you should filter out any *contact bounces* resulting from the mechanical nature of the button. To that end, an inter-arrival time less than 100 milliseconds between two subsequent calls to your button call-back method should be interpreted as a contact bounce, and the last method call should not be counted as a valid activation of the USER button.

IMPORTANT: Due to an MD407 board hardware design imperfection the CAN cable must be connected for the USER button activations to be detected correctly.

Problem 4.a: Demonstrate that your MD407 board software is able to measure the inter-arrival time between two subsequent valid activations of the USER button while in the PRESS-MOMENTARY mode. Make sure that the displayed inter-arrival time correspond well with the timing of your button-pressing pattern.

Problem 4.b: Demonstrate that your MD407 board software is able to enter and leave the PRESS-AND-HOLD mode. Make sure that the displayed time in seconds correspond well with the length of the time interval that the button remained pressed.

Assistant's approval: ....

[5]Calls to the button call-back method that are not caused by a contact bounce (see special note.)

[6]That is, from the time instant the button was pressed to the time instant the button was released.

Tap tempo:

To mimic one of the popular innovations for music equipment you should now implement a *tap tempo* functionality by means of the USER button on the MD407 board.

A *tempo-setting burst* is a sequence of four valid activations of the USER button, with the special property that the three inter-arrival times within the burst are of comparable length. Here, "comparable" means that, within the burst, no inter-arrival time differs from another inter-arrival time by more than 100 milliseconds. Once a tempo-setting burst is detected the average value of the three inter-arrival times within the burst should be used to define the tempo.

Extend the functionality of your button call-back method so that it is able to detect and analyse a tempo-setting burst, and print out[7] the corresponding tempo expressed as beats per minute (bpm). It is not a requirement to handle sequences with more than four valid activations of the USER button.

Integrate the tap tempo functionality in your music player, so that you now also have the option to set the tempo of the melody by means of the USER button.

The special requirements on the tap tempo functionality are as follows:

1. The music player should be able to handle a tap tempo change via the board's USER button while the melody is being played. By means of this functionality, it should be possible to set a `tempo` in the extended range of $[30 \ldots 300]$ bpm
2. The music player should be able to reset the tempo to its default value by means of a 2-second PRESS-AND-HOLD activation of the USER button, while the melody is being played. The default value for the music player is `tempo` $= 120$ bpm.

Problem 4.c: Demonstrate that your MD407 board software is able to set and reset the tempo in your music player by means of the USER button. Make sure that the software is able to handle any request for a new tempo setting in the extended range of $[30 \ldots 300]$ bpm. Also make sure that the software prints out the new tempo after a change (set or reset).

Assistant's approval: ....

Blinking LED:

On the MD407 board there is a green LED (marked 'LED2' on the board) that can be controlled with the `SIO` device driver that you have installed. To that end, the following LED functionality is available:

- With the operation `SIO_WRITE()` you are able to explicitly set the state of the green LED: a parameter value of 0 will turn the LED on ("lit"), while a parameter value of 1 will turn the LED off ("unlit").

[7]These printouts should replace the previously-used printouts of inter-arrival times.

- With the operation `SIO_TOGGLE()` you are able to invert the current state of the LED (from on to off, or vice versa).
- The default state of the green LED, after the device driver has been initialised, is turned on ("lit").

Integrate the blinking LED functionality in your music player, so that you now will be able to indicate the current tempo with a blinking LED.

The special requirements on the blinking LED functionality are as follows:

1. The green LED should blink with a frequency corresponding to the tempo (bpm). Thus, the time between each LED blink should be the length of a beat.
2. The LED should blink *fully synchronised* with the beat of the melody, meaning that the LED should be turned on ("lit") at the beginning of a beat and turned off ("unlit") in the middle of a beat.
3. After a tempo change the green LED should adapt its blinking frequency to indicate the new tempo, and continue to be fully synchronised with the melody. This applies regardless of whether the new tempo is set via the keyboard or the USER button.

Problem 4.d: Demonstrate that your MD407 board software is able to provide the blinking LED functionality in your music player. Make sure that the green LED always blinks at a frequency corresponding to the current tempo (bpm), fully synchronised with the beats of the melody.

Assistant's approval: ....

IMPORTANT! After approval you should submit your software code in Canvas (under 'Modules'/'Laboratory Software'). The code should be submitted as is, that is, in the shape it was at the time of approval (without modifications).

Create a '.zip' archive file, containing all your modified C files, and name the archive file '`Part2_4_PGID.zip`' (where `PGID` is your project group name). Typically, you only need to include the `application.c` file in the archive file, but if you have created additional files you should include them too.

A short `User's Guide`, describing how to control the software with the key commands, should be included (by means of C program comments or console print-outs) in `application.c`.

That's it for the laboratory assignment.
Thank You for the Music!