LSP580 - English
LET627 - Introduction to Real-Time Systems

Playing Brother Jacob: A Simple Task Made Hard With Computers

Writing Group 11

Andy Alavinasab, Sebastian Bergdahl, Mehdi Haidari and Emil Johansson

Chalmers
20-05-2022

Table of contents

Abstract

This report contains solutions and lessons relating to a laboratory assignment of playing “Brother John” using a built-in Digital-To-Analog converter. A basic overview of the project is introduced and the relevance of real-time systems is explained. Solutions to the design of the program are covered by one group and the implementation of the CAN protocol by the other. The Result of the assignment is a program capable of playing a simple song with the basic foundation for multi-card support.

1 Introduction

The purpose of this report is to illustrate the lessons learned and solutions used to solve problems given in a laboratory assignment using real-time systems. The task is to play the nursery rhyme “Brother John” by using a MD407 card. Solving such a task starts with an understanding of how we create sound.

Sound can be created by using a built-in digital-to-analog converter (DAC), on a MD407 card. By writing alternating values to the DAC a tone can be created. A different tone can be played by changing the frequency of these alterations.

1.1 Task description

To control the tone user input is needed. The use of real-time systems programming allows the user to enter inputs mid-tone without interrupting the tone. With this basic function, further developments can be made to allow for muting, stopping, starting, and changing key, tempo, or volume.

If a heavy process overlaps with the frequency it can create distortion. To explore the concept a background load is run simultaneously to the program and an audible difference can be heard at heavier loads. Using deadlines the program can be future-proofed against long processes and always give a clean tone. After the effect has been studied and understood there is no reason to keep the background load.

Using a table of pre-calculated frequencies, a tone can be transformed into a song. The frequency table uses preprogrammed indices to play a chosen song. These indices shift depending on which key is selected. A key is of no use unless the program knows for how long to play every tone so a second process must be added. The second process keeps track of how long each tone needs to be played and is the reason real-time programming is used to solve the task.

The length of a tone can be expressed in the length of a beat. The beat is determined by the tempo to create an index of the tone lengths and a key is used to shift the index. After the additional user inputs are added allowing the tempo and key to be changed the program is ready to play a song. To start the song a play and pause button can be implemented to allow further control over the user's experience.

The aim is to create a program capable of producing a simple song using the above-mentioned process to learn how to use real-time systems, the Tinytimber kernel, and the external MD407 card.

1.2 Technical Theory

The MD407 card used is an ARM-based microcomputer. It takes advantage of a 32-bit ARM processor labelled SMT32F407. The card has been developed for laboratory use at Chalmers University. The main purpose of the card was to be used for machine-oriented programming but also has capabilities for use in real-time systems with the CAN bus[1].

Tinytimer is a real-time kernel used for programming applications using real-time systems. It utilises deadlines and priorities to allow multiple tasks to run concurrently and allows for tight time controls. Utilising real-time systems with timing controls is a requirement for many modern technologies such as an aircraft or a loudspeaker.

To enable multi-card support data needs to be able to transfer between connected cards. This can be done using the Controller Area Network (CAN) bus. Data can be sent from a card using the CAN bus which can then be received by a connected card. The sender can also be connected to itself, thus responding to its own CAN messages.

CAN messages generate an interrupt once received and the data contained inside the message can be interpreted. The data inside the CAN message is stored in the “data field” of the CAN message which consists of a maximum of 8 bytes. Meaning that the CAN message data can at maximum consist of a char array with 8 positions [2].

When multiple cards interact they are labelled according to their purpose. The cards that are sending out data are labelled leader cards and the cards responding to the data are labelled salve cards. An important note is that a card can switch mode of operation depending on the circumstance.

An acknowledgement is sent from the slave card to the leader card confirming a successful checksum check. The checksum is responsible for validating the integrity of the data inside the CAN message. If the leader card does not receive a dominant level output (an acknowledgment) it detects an acknowledgment error [2].

When A damaged frame is identified and labelled by the leader card. Any defective frames get aborted and retransmitted according to the CAN recovery procedure. If multiple messages are sent during a window where errors are generated, such as if the bus is disconnected, a clash occurs. During a clash, the CAN bus provides a non-destructive bus arbitration and all messages are sent according to their assigned priority[2].

2 Technical solution

Building a program like this usually starts with the software and hardware selection. The selection to use a MD407 card and the TinyTimber kernel was already made by the task instructions. Alternatives exist such as the ada95 software and a multitude of different cards, but the tasks given were designed for the chosen card and software.

2.1 Music player

When designing the program it was decided to use three different objects: a reader, a tone generator, and a controller. The program is able to be run using only 2 objects which was less efficient for the processor, so it was optimised by using 3 objects instead.

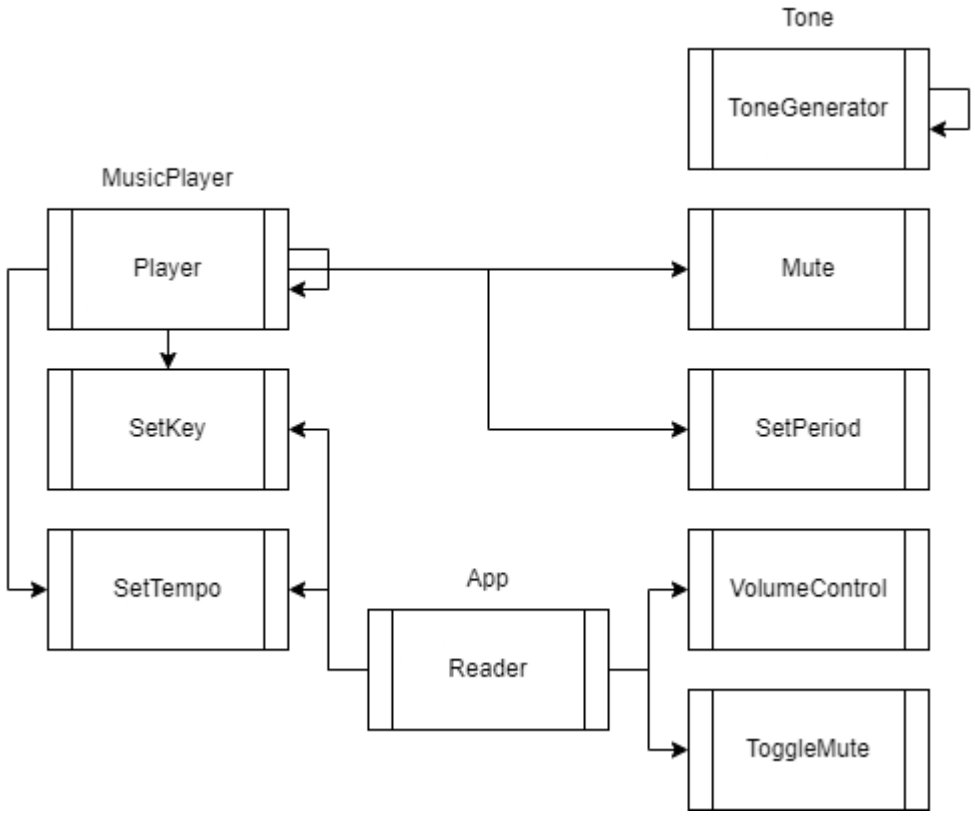

Figure 1. Access graph showing interactions between objects and their functions.

The reader function handles and interprets all the user input generated from the keyboard and calls different functions depending on which input is received as demonstrated in figure 1. Because the reader function is an interrupt it can only be called when an interrupt is received. The reader also uses the App object which stores relevant information to the reader such as a local message buffer. This buffer enables the reader to receive input values with multiple characters such as negative numbers.

The first function added is a volume controller. A user can increase or decrease the volume by pressing the “+” and “-” keys respectively. The reader function interprets what key has been pressed and then adds or subtracts 1 from the value of the volume variable inside the Tone object as seen in figure 1. The volume has been constrained to not go above 20 for safety reasons and is limited to above 1 as a separate mute function exists.

There are two separate mute methods that serve different purposes. As seen in figure 1 the mute method is called by the player in order to create a gap between notes. The toggleMute-function allows the user to mute the tone generator through a flag that controls the previously mentioned mute. Muting the player can be done with the ‘m’ key and it always sets the volume sent to the DAC 0 no matter how the volume variable is increased or decreased. The ‘m’ key is designed to be a toggle and if the program is already muted pressing the key unmutes it.

Changing the volume and muting is only a part of the user control needed to control the player. A user can enter several keystrokes that are saved in a buffer. The buffer is then interpreted to see if one of the keys “k” or “t” is pressed, The keys represent a change to the key or tempo respectively. This is necessary for the user to enter negative numbers to control the key or 2 to 3-digit numbers to control the tempo. Before entering any instructions the user must press the ‘x’ key to clear the buffer.

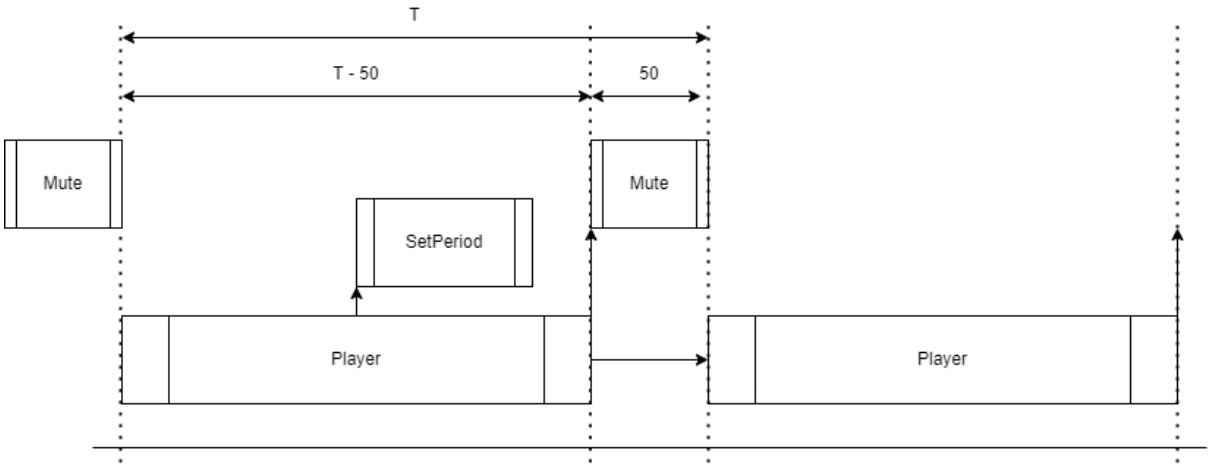

Figure 2. A timing diagram showing chosen delays between notes.

As illustrated in figure 2, each note has the length of T. T is divided up into two parts: the note and the gap. The length of T varies depending on the current tempo of the song but is by default set to half a second. So the note is then played for 450 milliseconds and then a gap of 50 milliseconds is given before the next note is played. The tempo of the song can be changed dynamically as described above to either lengthen or shorten these timings. The gap, however, is constant.

To play a song notes are needed which are generated by the tone object. A tone is continually generated by the tone generator method. By recursively calling itself, the method generates a tone by alternating between setting the DAC to the value of the volume and zero. Alternating the DAC in this way creates a wave which in turn generates a sound. The note is decided by what rate the function calls itself (period) and to create a song, different notes are required. However, this note is constant, and to change the period needs to be changed.

The calculation for which period to update the tone object is done by the MusicPlayer object, mainly by the player function. Just like the tone generator, the player also recursively calls itself. How often the method does this differs depending on which iteration it is going through. Every time the method is called, a counter is increased. Depending on what value the counter has, different things occur. Firstly, the calculations to decide both the Beats Per Minute (BPM) of the song and the period of the tone generator are done by getting a value from an array of saved multiples and periods respectively decided by the current value of the counter is. Secondly, if the current value of the counter is even, the player method updates the period in the tone object and then also calls on itself after a delay decided by the current BPM. On the other hand, if the value in the counter is odd, the player simply calls itself after a delay of 50 milliseconds. Finally, every time the method is run, the mute function is called in order to set the volume of the tone to 0. The reason for this is to create gaps between the notes.

For the program to know what delay it should have it reads from the frequency index. The index contains set frequencies that correspond to notes in the range of [247, 988] Hz. The order of what notes are to be played is kept in a list and can be shifted by changing the key, with the setKey function. The function shifts every entry in the list by an amount entered by the user and is limited by the range of frequencies in the program.

2.1 CAN support

Finally, to start the program, the play button is implemented, allowing the user to start with the ‘p’ key. This keypress starts both the tone generator and the player methods. The function has unintended interactions when a card is swapped rapidly between slave and leader functions. These interactions differ from each user implementation and can cause the song to be played twice out of sync at the same time, no song to play at all, or that no notable change happens.

To allow the program to be run in either leader or slave mode a new method of sending instructions is implemented. The CAN function allows the program to receive instructions from other cards allowing it to run as a slave and also send instructions along the CAN bus. The way it is tested on the card is by attaching a network cable from the output to the input and having the card send instructions to itself.

The CAN messages that are sent are identical to the keyboard controls with one exception. If the user enters one or several keys followed by the ‘e’ key the message gets changed. Instead of sending everything at the same position in the message the letter ‘e’ gets sent in the first position. Following the ‘e’ is every key in the order they were pressed, examples being ’-’, ‘5’ and ‘e’, in that order becomes ‘e’, ‘-’ and ‘5’. The change of order allows the program to first check if it was sent a sequence of keys and then read the contents of the following cells in the message.

3 Results

The player is able to play the song outlined in the guidelines and an additional song is implemented in a different program file. The quality and implementation of the second song are faulty due to lacking knowledge of reading sheet music but is still playable. Further development to allow for playing chords and adapting volume to simulate instruments were explored but quickly abandoned because of a lack of time and knowledge.

To adapt the DAC to emulate instruments the volume written needs to be dynamic and change within every tone. To play the second song as originally composed requires pizzicato using a violin which creates a high volume that quickly dampens. This dampening function is lacking in the current program but could with further development be added.

3.1 Discussion

From the start the project has been designed with the purpose of learning, not to be an optimised end product. To design the program to be an end product would require fundamentally changing the process used to reach the end goal. This means that a lot of polishing is left to do and many parts would have to be rebuilt from the ground up.

The program is built in iterations and functions are added successively. This design method leads to compounding issues and causes the program to be less optimal than possible. The current design causes no issues but further developments would need to take the compounding issues into heavy consideration. Flaws such as the lack of scalability are also a major issue when adapting the program to different songs or tempos outside of the originally intended range.

Many of the current issues would be avoided by starting the project with a clear end goal and a map of what parts are needed. Designing this map would require the skills that are intended to be learned during the project and would only be a reasonable implementation if a new attempt was made to make an optimal design.

Our first source consists of one article about the MD407 card, detailing the specifications and functionality of the card. It is deemed a highly reliable source since it’s published and verified by the Chalmers institute of technology.

The second source is a book by Wilfried Voss detailing the functions of the CAN protocol. As for this source, it is not guaranteed that all information is valid, but the information we used corresponds to what has been taught in the course.

Wilfred Voss is the owner of the Copperhill Technologies Corporation that published the book cited. Copperhill Technologies Corporation sells CAN bus technology to other corporations. Hence it is in his interest to not highlight potential issues with the technology. Therefore the use of the technology should not solely rely on statements from this book if required for safety-critical use.

3.2 Conclusion

In conclusion, the project's goal was to teach how to build a program utilising real-time systems from scratch. Most of the issues encountered along the way were in part due to lacking knowledge, surrounding the subject, or the lack of planning. The major lesson to be learned is that proper planning is required, from start to finish, in order to have a good structure and a clear goal. This would lead to fewer issues, and greater scalability but also contribute to easier-to-understand code. With better planning, methods would be added to contributing to the greater goal in mind rather than to fixing a short-term problem.

4 Sources

[1] F. Östman, “MD407 – En ARM-baserad laborationsdator för utbildning,” *odr.chalmers.se*, 2016, Accessed: Jun. 11, 2022. [Online]. Available: https://hdl.handle.net/20.500.12380/232582

[2]W. Voss, *A comprehensible guide to Controller Area Network*, 2nd ed. Greenfield, Mass.: Copperhill Technologies Corp., 2008.