Personal Trainer Report

Report for the developing of Audio Feedback with stm32f4discovery - GROUP 31

Calcina Tommaso
Matr. 817711, (tommaso.calcina@gmail.com)
Garetti Filippo
Matr. 815810, (pippogarett@yahoo.it)
Mezzanotte Marco
Matr. 816479, (mezzanotte.marco@gmail.com)

Report for the master course of Embedded Systems
Reviser: PhD. Patrick Bellasi (bellasi@elet.polimi.it)

January, 28 2014

Abstract

The main goal of our project was to give an audio feedback for some of the main features of the Personal trainer application. For example make the board play audio samples to reproduce the exact number of walks done, or play a victory or loss sound when the sensor catches a number of walks lower or greater from another board.
To make this happen we developed some functions that can be called from other modules passing the correct parameters. Interfaces for these functions can be found in a header file.

1 Introduction

The very first step of our project has been analyze and understand how to use or better customize the player already implemented in the examples folder in MiosixOS. Then we tried different sampling dimensions to know how much memory this would have cost.
The second step consisted in record and sample all audio files we needed for the feedback. We tried different dimensions for the sampling and after a little study of the trade-off between quality and usage of memory we chose to set the frequency to 8kHz in 16 bit, in order to occupy more or less 380KB in the end. The final big step was to integrate our functions and the customized player in order to play the different sounds requested.

1.1 Analyze and customize the player

In the beginning of our project we studied how to better re-use (or customize) the class player.cpp already writtend in the examples folder of MiosixOS. We understood that in order to reproduce any sound it's necessary to enable the DMA, and the Serial Bus SPI to send audio samples from the memory to the Digital Analog Converter. The DAC has to be initialized and it's also necessary to set the established reproduction frequency (8kHz) of the PLL Audio.
Initially we used the `player.cpp` as it was, only modifying the frequency of the PLL Audio. But, after the improving of the development of the project, we used the player.cpp as a base for the implementing of our `play_V` function in the same file (`player.cpp`). We needed to implement the empty constructor of the ADPCMSound class, in the same file, so that we could create the ADPCMSound array without initialize it. In fact everytime is needed to play an audio sample, we retrieve the associated binary file (ADPCM encoded array from a .h file) and the associated length.

1.2 Recording and Sampling Audio Files

We recorded all audio files with the tool *Audacity* setting also here the sampling frequency to 8kHZ in mode 16bit PCM Mono. All the choices were made in order to occupy the least memory as possible. We used the CODEC and the Converter given, again, in the examples folder in MiosixOS. We obtained all the .h files containing all the arrays of characters that have to be used by the `slice-and-play.cpp` module.

1.3 Integrations between Functions

After the development of the 2 slice-and-play.cpp function we imported the player and the adpcm in our function. We imported also all the audio files using only one file .h containing them all. Using the sound class we create objects that the player can use to reproduce some audio.
`someFile.h -> slice-and-play.cpp` asks -> `adpcm.c` responds to `slice-and-play.cpp => player.cpp`
Our module is based on calls from other modules. In order

to get to this, we created the .h interface that declares the `ring` class that provides all the public functionalities of the `slice-and-play.cpp`.

2 slice-and-play.cpp

This class is the core of our project, contains the algorithm that can identify all the audio files representing the number in its integrity, it can play these audio files and run two functions that can play a victory or a loss sound. Starting from the integer, the algorithm divides it by a million, a hundred thousands, ten thousands and so on until the number one, takes the results of every operation and using the exponents identifies the audio file that has to be reproduced each time.
For example, a loop consists in these fundamental operations:

57 => entering in the loop
...(divisions which return zero - no actions)...
step of tens: 57-($10^2 * 0$)/10=5 exp = 1 => `saytens(5);`
step of units: 57-($10^1 * 5$)/1= 7 exp = 0 => `sayunits(7);`

3 Integration among modules

Our module was developed to easily integrate with other modules, so that once the headers of our functions were put in the interface the other groups could call our functions just including the .h file.
To better exploit the player functionalities, together with other groups, we suggested to create a reserved thread containing the call (or the calls) to our functions.
We integrated our module with wifi group and pedometer group.