

Sprawozdanie

„Odtwarzacz plików dźwiękowych na STM32F407 (płytka STM32F4 Discovery)”



Autorzy:

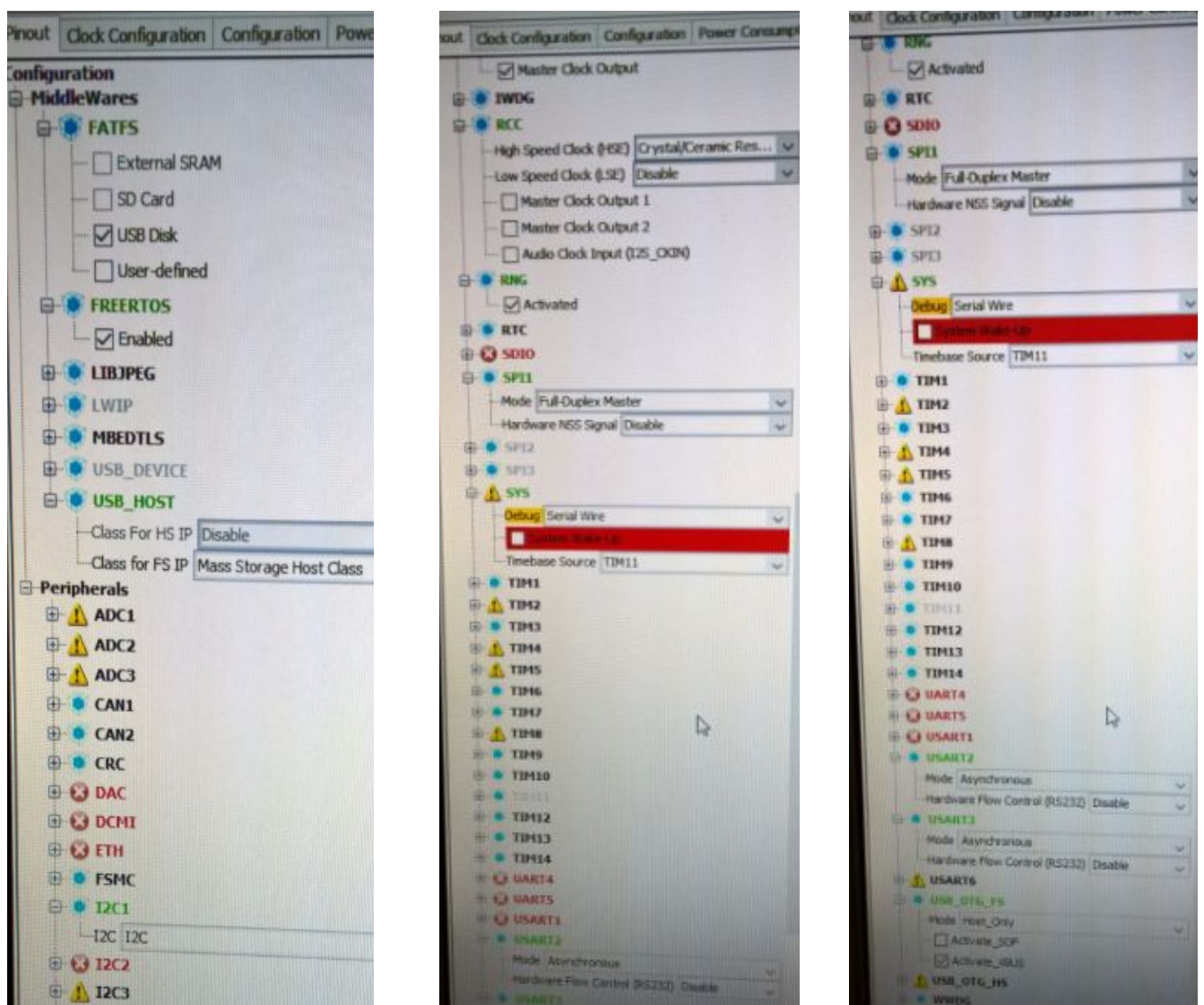
Klaudia Knafel

Agata Bogacz

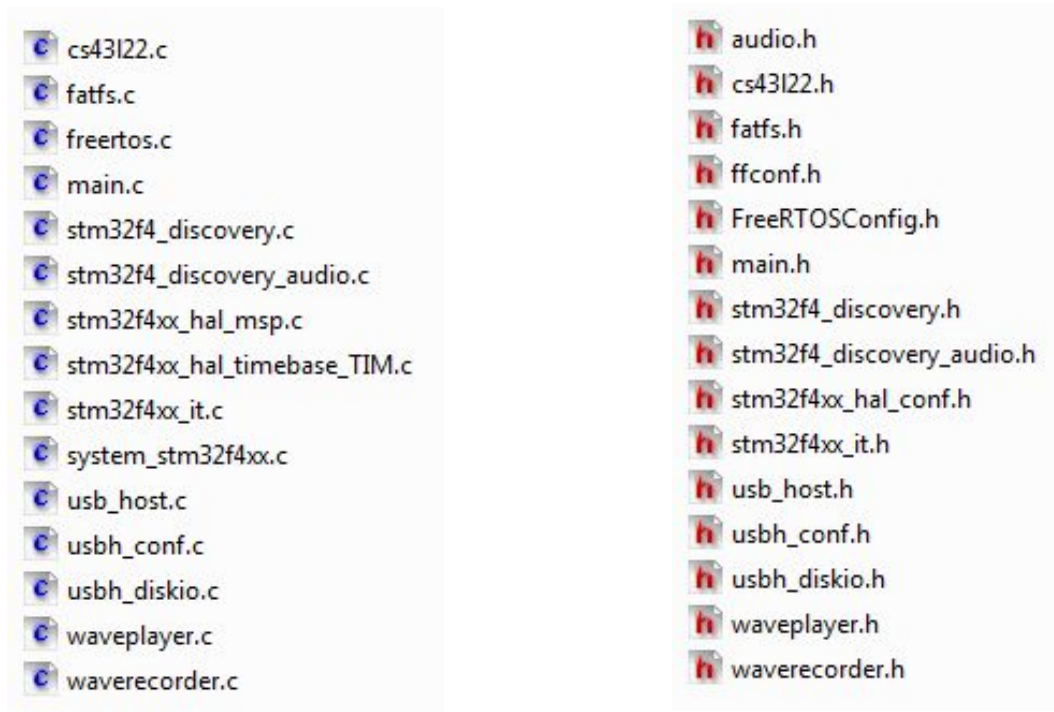
Celem zadania inżynierskiego było zaprogramowanie płytki STM32F4 Discovery tak, aby odtwarzała pliki z rozszerzeniem MP3. Niestety, nie zdążyliśmy doprowadzić do odtworzenia pliku o rozszerzeniu MP3, za to implementacja podstawowej funkcjonalności polegającej na odtwarzaniu nieskompresowanych plików .wav zakończyła się powodzeniem. Wykonanie zadania składało się z kilku etapów, które przedstawiono poniżej.

1. Wygenerowanie kodu w STM32 CubeMX.

W tej części zadania należało ustawić odpowiednie opcje i wygenerować Makefile, na podstawie którego potem został utworzony projekt w środowisku Eclipse:



Po wygenerowaniu kodu dodałyśmy do katalogu Src i Inc odpowiednie pliki z folderu z bibliotekami STM32 (po wcześniejszej aktualizacji wersji programu STM32 CubeMX). Ostatecznie, zawartość Src i Inc wyglądała następująco:



Plik *waveplayer* został wykorzystany w dalszej części projektu do napisania uproszczonej wersji własnego odtwarzacza plików typu *wave*. Ponieważ odtwarzacz miał służyć do odtwarzania plików z urządzenia USB konieczne było dodanie biblioteki *usb_host*. Dzięki bibliotece *fatfs* można było dokonać inicjalizacji i zamontowania systemu plików z urządzenia USB.

W dalszej części należało dodatkowo załączyć pliki biblioteki Helix aby móc odtwarzać nie tylko pliki wav, ale również MP3.

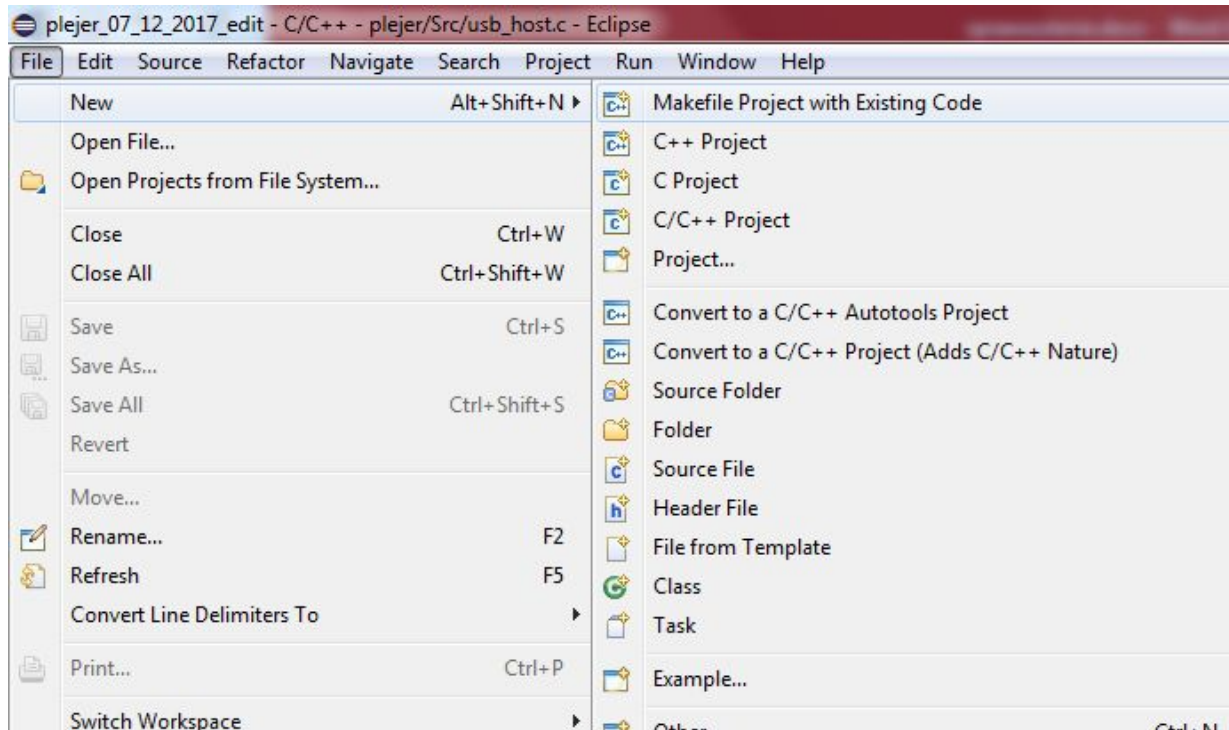
W celu dołączenia plików, np. *stm32f4_discovery_audio.h* trzeba było dodać w pliku main instrukcję `#include`, w tym przypadku:

```
#include "stm32f4_discovery_audio.h"
```

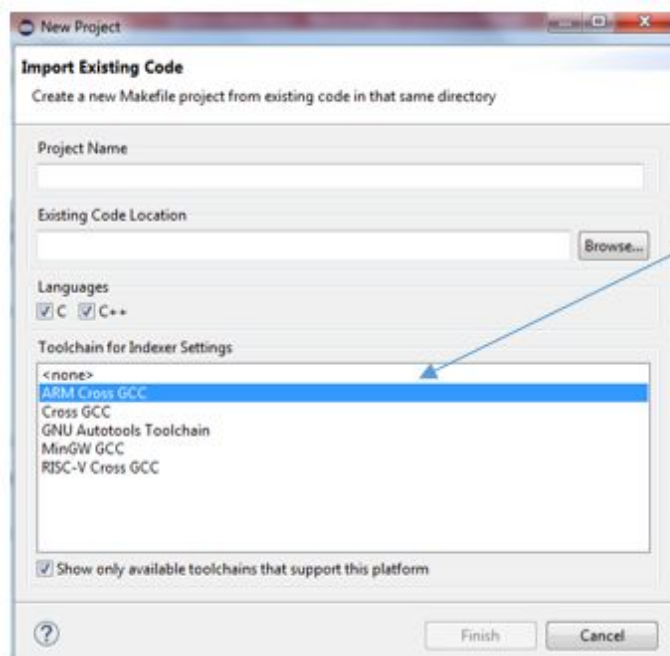
zwracając szczególną uwagę na to, żeby nie dołączać pomyłkowo w ten sposób pliku z rozszerzeniem `.c`...

2. Wstępna kompilacja za pomocą środowiska Eclipse.

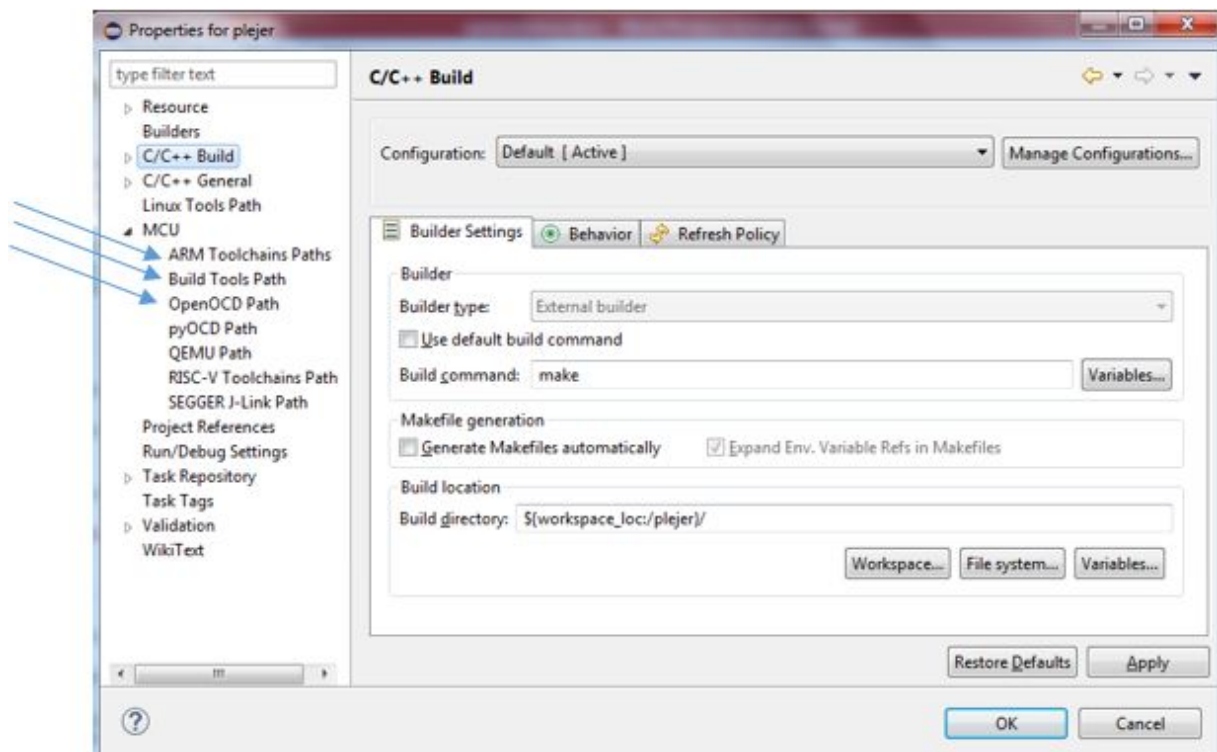
W celu uruchomienia szkieletu programu użyliśmy programu Eclipse Neon z pakietem oprogramowania GNU MCU Eclipse. Na początku utworzyliśmy „Makefile Project with Existing Code”:



Następnie wybrałyśmy opcję ARM Cross GCC:



Po utworzeniu projektu należało ustawić „Build command” na „make”:



W miejscach wskazywanych przez strzałki należało wpisać, w przypadku komputerów w pracowni, odpowiednie ścieżki:

ARM Toolchains Path: C:\gcc_toolchain\6_2017-q1-update\bin

Build Tools Path: C:\gcc_toolchain\Build_Tools\bin\bin

OpenOCD Path: C:\gcc_toolchain\openocd-0.10.0\bin-x64,

z nazwą pliku wykonywalnego **openocd.exe**.

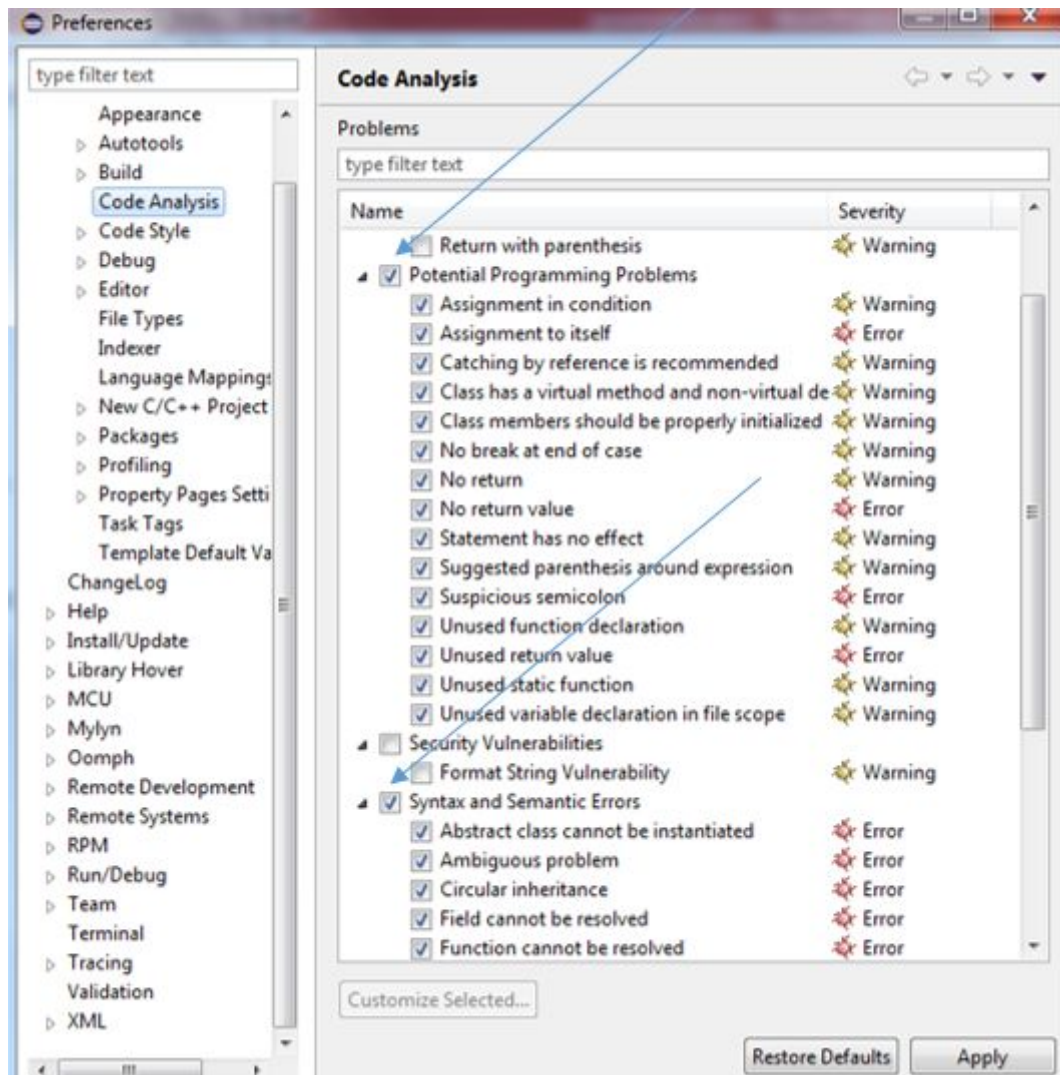
W pliku makefile należało ustawić dodatkowo makro

BINPATH = "C:\gcc_toolchain\6_2017-q1-update\bin".

Po wprowadzonych zmianach projekt był gotowy do kompilacji.

Pomimo wykonania poprawnie powyższych kroków napotkaliśmy wiele komplikacji związanych z działaniem Eclipse. Przy konfiguracji na własnym komputerze należało zwrócić uwagę na to, że wyświetlane komunikaty dotyczące błędów nie zawsze były trafne ze względu na niedopracowanie środowiska Eclipse. Rozwiązaniem problemu ataku tysiącami wyrzucanych bezpodstawnie błędów zwanych „Codan Errors” było wyłączenie

ich zgłaszania. Można to wykonać wchodząc w *Window -> Preferences -> C/C++ -> Code Analysis*:



i odznaczając *Potential Programming Problems* i *Syntax and Semantic Errors*.


Przydatnym narzędziem przy pracy ze środowiskiem Eclipse okazała się opcja

Clean Project. Często, gdy wydawało się, że w programie jest błąd wystarczyło użycie tego przycisku i problem sam się rozwiązywał.

3. Uruchomienie narzędzia do debugowania i wgrywania kodu wynikowego.

W poprzednim punkcie przedstawiono w jaki sposób ustawić ścieżkę do OpenOCD. Pełna nazwa narzędzia to Open On-Chip Debugger. Służy on do debugowania i wgrywania kodu na płytke. Z tego też względu w tym punkcie należało podłączyć płytkę STM32F4 Discovery do komputera.

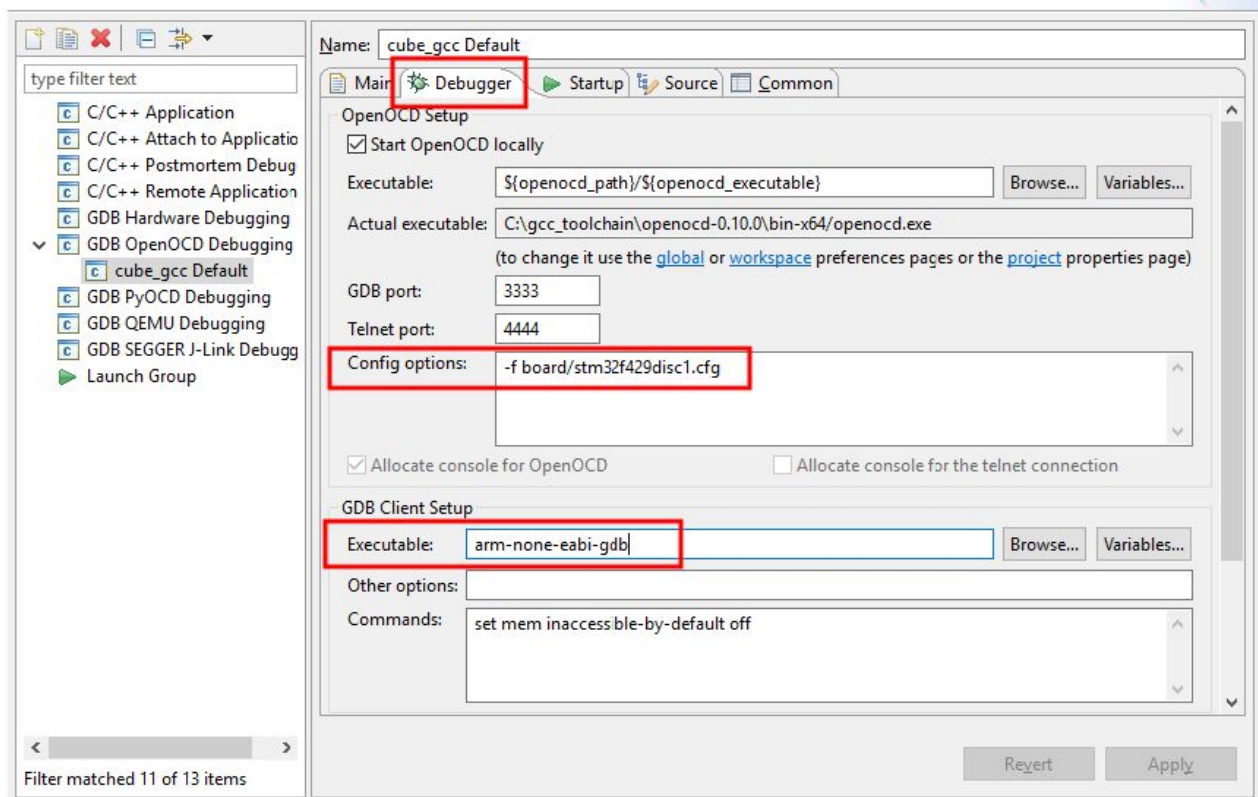
Po podłączeniu urządzenia do portu USB należało w środowisku Eclipse wybrać
Run -> Debug Configurations -> GDB OpenOCD Debugging

i następnie kliknąć na pole  oznaczające tworzenie nowej konfiguracji.

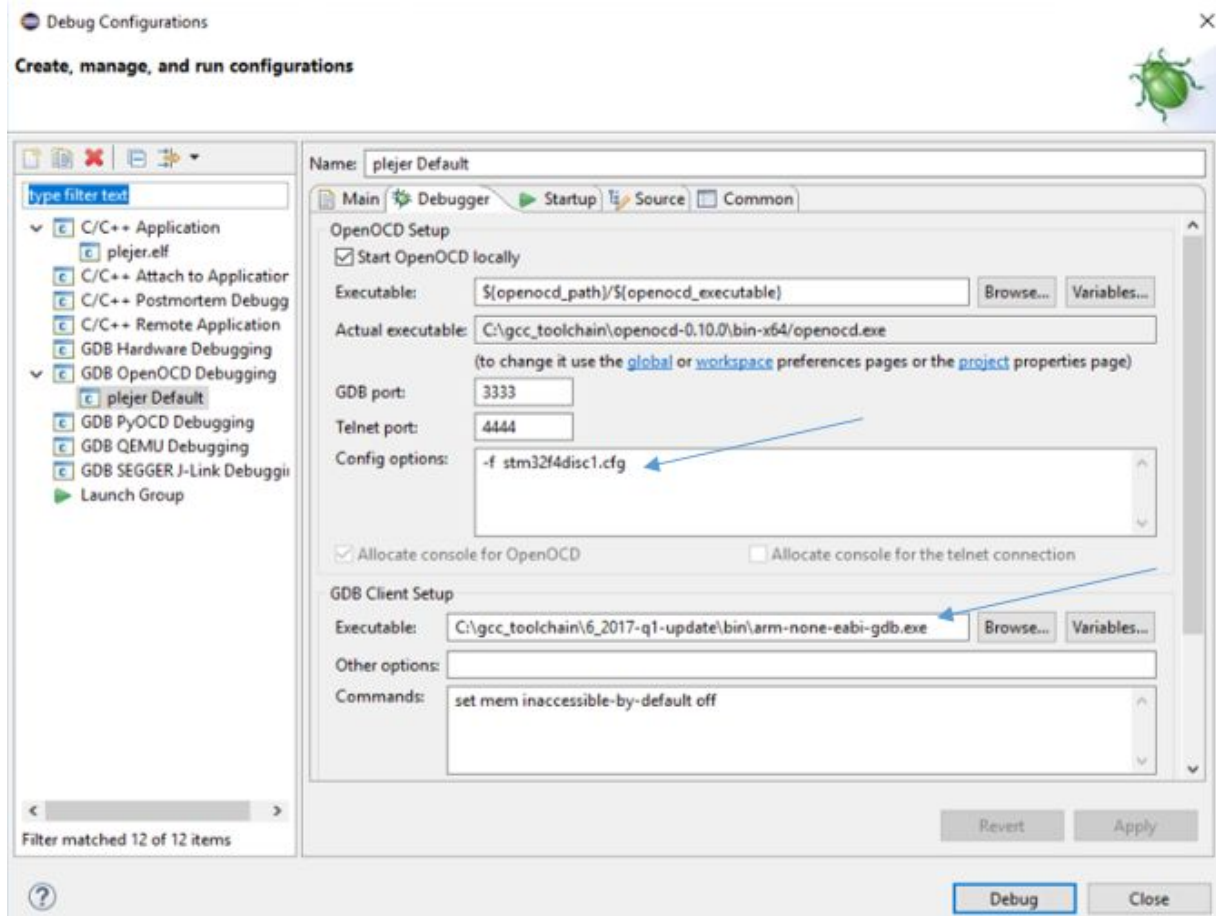
Skonfigurowanie debuggera okazało się dość problematyczne.

Oto ustawienia przedstawione w instrukcji do jednego z ćwiczeń laboratoryjnych, które wykonaliśmy w trakcie pierwszej części laboratoriów w bieżącym semestrze:

Create, manage, and run configurations



Niestety, okazało się, że Eclipse nie może znaleźć pliku konfiguracyjnego dla wybranej płytki testowej. Konieczne było ręczne wskazanie dokładnej ścieżki do gdb oraz pewne dalsze zmiany. Oto ustawienia zastosowane na komputerze w laboratorium:



Pola oznaczone strzałkami należało wypełnić tak jak na obrazku, odnaleźć plik *stm32f4discovery.cfg*z *\openocd-0.10.0\scripts\board*, skopiować, wkleić do katalogu zawierającego projekt i zmodyfikować linijkę:

```
source [find interface/stlink-v2.cfg]
```

dodając -1

```
source [find interface/stlink-v2-1.cfg].
```

Następnie należało zmienić nazwę pliku na *stm32f4disc1.cfg*. Po tych krokach możliwe było uruchomienie debugera.

4. Dalsze modyfikacje kodu.

- Dodanie funkcji callback z pliku *waveplayer.c* do *main.c*.

```
void BSP_AUDIO_OUT_HalfTransfer_CallBack(void)
{
    buffer_offset = 1;
}
```

```
void BSP_AUDIO_OUT_TransferComplete_CallBack(void)
{
    buffer_offset = 2;
    BSP_AUDIO_OUT_ChangeBuffer((uint16_t*)&Audio_Buffer[0], FILE_READ_BUFFER_SIZE /
2);
}
```

- Dodanie do *main.c* funkcji, dzięki którym można było używać *printf*. Podczas testowania działania programu i jego debuggowania, znacznym ułatwieniem było wykorzystanie wypisywania komunikatów na ekran konsoli.

```
static void print_chr(char chr)
{
    HAL_UART_Transmit(&huart2,
        (uint8_t*)&chr, 1, 1000);
}
```

```
ssize_t _write_r(struct _reent *r, int fd, const void *ptr, size_t len)
{
    int cntr = len;
    char *pTemp = (char*)ptr;
    while(cntr--)
        print_chr(*pTemp++);
    return len;
}
```

Przy korzystaniu z funkcji *printf* wypisywany komunikat należało zakończyć znakiem końca linii, np.

```
printf("init done\n");
```

- Dodanie inicjalizacji hosta USB na płytce wykorzystując funkcję z załączonego wcześniej pliku `usb_host.c`:

```
MX_USB_HOST_Init();  
osDelay(1000);
```

Oczekiwanie na gotowość aplikacji przed kolejnymi krokami programu:

```
while(Appli_state != APPLICATION_READY)  
    osDelay(1);
```

- odczytywanie plików:

Poniżej przedstawiono kroki podjęte w celu odczytania pliku, tutaj na przykładzie pliku z rozszerzeniem `.wav`.

```
char* filename = "test.wav";
```

Za pomocą funkcji `f_mount` zarejestrowanie systemu plików do modułu FatFs (zainicjowany wcześniej poleceniem `MX_FATFS_Init();`):

```
static FATFS fs;  
f_mount(&fs, "", 1);
```

W celu otwarcia pliku należało zadeklarować desktyptor pliku oraz otworzyć go za pomocą funkcji `f_open`:

```
FIL FileRead;  
FRESULT f1 = f_open(&FileRead, filename, FA_READ);
```

Zwrócone z funkcji `"FR_OK"` oznaczało poprawnie otworzony plik.

```
If (f1 == FR_OK)  
{  
    f_read (&FileRead, &waveformat, sizeof(waveformat), &bytesread);  
    ...  
    f_close(&FileRead);  
}
```

`f_read` – odczyt pliku, `f_close` – zamknięcie pliku po zakończeniu odczytu.

- dodanie funkcji default handler do celów debugowania programu
- Tutaj kolejny raz napotkaliśmy trudności z debugowaniem programu z powodu default handlera (początkowo ustawiony na po prostu nieskończoną pustą pętlę) uruchamiającego się w przypadku wystąpienia wyjątku, dla którego nie została napisana funkcja jego obsługi.

W pliku *stm32f4xx_it.c* należało dodać następujące instrukcje:

```
void Default_Handler_HL(void)
{
    printf("to jest default handler!!\n");
    while(1);
}

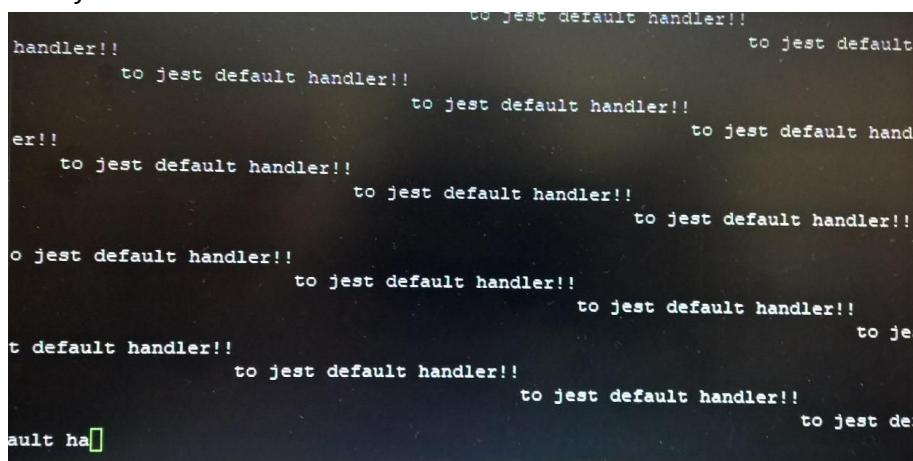
extern I2S_HandleTypeDef hAudioOutI2s;

void DMA1_Stream7_IRQHandler(void)
{
    HAL_DMA_IRQHandler(hAudioOutI2s.hdmatx);
}
```

A w pliku *startup_stm32f407xx.s*:

```
Default_Handler:
Infinite_Loop:
    b Default_Handler_HL
b Infinite_Loop
```

Po podłączeniu się poprzez putty (w naszym przypadku zaznaczamy serial, speed 115000 i COM5) w konsoli pojawiła się nieskończona pętla wypisująca „to jest default handler!!”, dzięki czemu wiedzieliśmy, że nasz program napotkał jakiś błąd i przeszedł do funkcji default handlera.



00:07:49 Connected SERIAL 115200 8 N 1

5. Odtworzenie pliku w formacie .wav

Nasz odtwarzacz wav oparł się na funkcji z przykładowego pliku odtwarzacza waveplayer.c. W pętli wczytujemy do bufora kolejne fragmenty pliku do odtwarzania, przy czym kolejna porcja jest wczytywana, gdy pierwsza część bufora zostanie już odtworzona (w jej miejsce) i analogicznie z drugą połową bufora.

Do otwierania pliku wav skopiowaliśmy strukturę z przykładowego projektu odtwarzacza do naszego pliku main.h:

```
typedef struct
{
    uint32_t ChunkID;    /* 0 */
    uint32_t FileSize;    /* 4 */
    uint32_t FileFormat; /* 8 */
    uint32_t SubChunk1ID; /* 12 */
    uint32_t SubChunk1Size; /* 16 */
    uint16_t AudioFormat; /* 20 */
    uint16_t NbrChannels; /* 22 */
    uint32_t SampleRate; /* 24 */

    uint32_t ByteRate; /* 28 */
    uint16_t BlockAlign; /* 32 */
    uint16_t BitPerSample; /* 34 */
    uint32_t SubChunk2ID; /* 36 */
    uint32_t SubChunk2Size; /* 40 */

}WAVE_FormatTypeDef;
```

Zadeklarowaliśmy zmienne potrzebne do odczytania i odtworzenia pliku wav:

```
static FATFS fs;
const int FILE_READ_BUFFER_SIZE = 8192;
int buffer_offset = 0;
uint8_t Audio_Buffer[8192];
WAVE_FormatTypeDef waveformat;
static uint32_t WaveLength;
static uint32_t AudioRemSize = 0;
```

Modyfikujemy główną funkcję:

```
void StartDefaultTask(void const * argument)
{
    /* init code for USB_HOST */
    MX_USB_HOST_Init();
```



```

osDelay(1000);

/* init code for FATFS */
MX_FATFS_Init();

while(Appli_state != APPLICATION_READY)    //oczekujemy na gotowość
    osDelay(1);

f_mount(&fs, "", 1);                        //montujemy system plików

/* USER CODE BEGIN 5 */
printf("Application ready!\n");
printf("init done\n");

FIL FileRead;                              //deklarujemy dekryptor pliku
UINT bytesread = 0;
char* filename = "test.wav";               //test.wav to plik, który chcemy odtworzyć

FRESULT f1 = f_open(&FileRead, filename, FA_READ);    //otwieramy plik
printf("%d\n", f1);    //na konsoli wyświetlamy zwróconą wartość z f_open, powinna wynosić 0

if(f1 == FR_OK)                            //jeśli plik został otwarty prawidłowo
{
    //odczytujemy porcję danych z pliku
    f_read (&FileRead, &waveformat, sizeof(waveformat), &bytesread);
    //w zmiennej WaveLength zapisujemy rozmiar pliku, który mamy odtworzyć
    WaveLength = waveformat.FileSize;
    //70 oznacza głośność odtwarzanego pliku, Sample Rate to częstotliwość odczytana
    z pliku wav
    BSP_AUDIO_OUT_Init(OUTPUT_DEVICE_AUTO, 70, waveformat.SampleRate);
    //przenosimy wskaźnik na początek pliku
    f_lseek(&FileRead, 0);
    //odczytujemy do bufora porcję danych
    f_read (&FileRead, &Audio_Buffer[0], FILE_READ_BUFFER_SIZE, &bytesread);
    //zaznaczamy ile danych mamy jeszcze odczytać
    AudioRemSize = WaveLength - bytesread;
    printf("file read\n");
    //odtworzymy odczytaną porcję danych
    BSP_AUDIO_OUT_Play((uint16_t*)&Audio_Buffer[0],
    FILE_READ_BUFFER_SIZE);
    //odtworzymy pozostałe fragmenty pliku
    while(AudioRemSize != 0)
    {
        bytesread = 0;
    }
}

```

//jeśli zostanie wywołany callback BSP_AUDIO_OUT_HalfTransfer_Callback to buffer_offset == 1, znaczy to, że pierwsza część bufora została już odtworzona i można do niej wczytać nową porcję danych

```
    if (buffer_offset == 1)
    {
        //czytamy kolejną porcję danych
        f_read (&FileRead, &Audio_Buffer[0],
FILE_READ_BUFFER_SIZE/2, &bytesread);
        buffer_offset = 0;
    }
//jeśli zostanie wywołany callback BSP_AUDIO_OUT_TransferComplete_Callback to wtedy buffer_offset == 2, co oznacza, że zawartość bufora została już odtworzona (callback wysłał już dane na interfejs)
    else if(buffer_offset == 2)
    {
        //wczytanie nowych danych do drugiej części bufora
        f_read (&FileRead, &Audio_Buffer[FILE_READ_BUFFER_SIZE/2],
FILE_READ_BUFFER_SIZE/2, &bytesread);
        buffer_offset = 0;
    }

    if(AudioRemSize > (FILE_READ_BUFFER_SIZE / 2))
        AudioRemSize -= bytesread;
//jeśli danych do odczytu jest mniej niż połowa pojemności bufora - kończymy pętlę
    else
        AudioRemSize = 0;
}
printf("End of file\n");
AudioRemSize = 0;
f_close(&FileRead); //zamykamy plik
}
```

/ Infinite loop */*

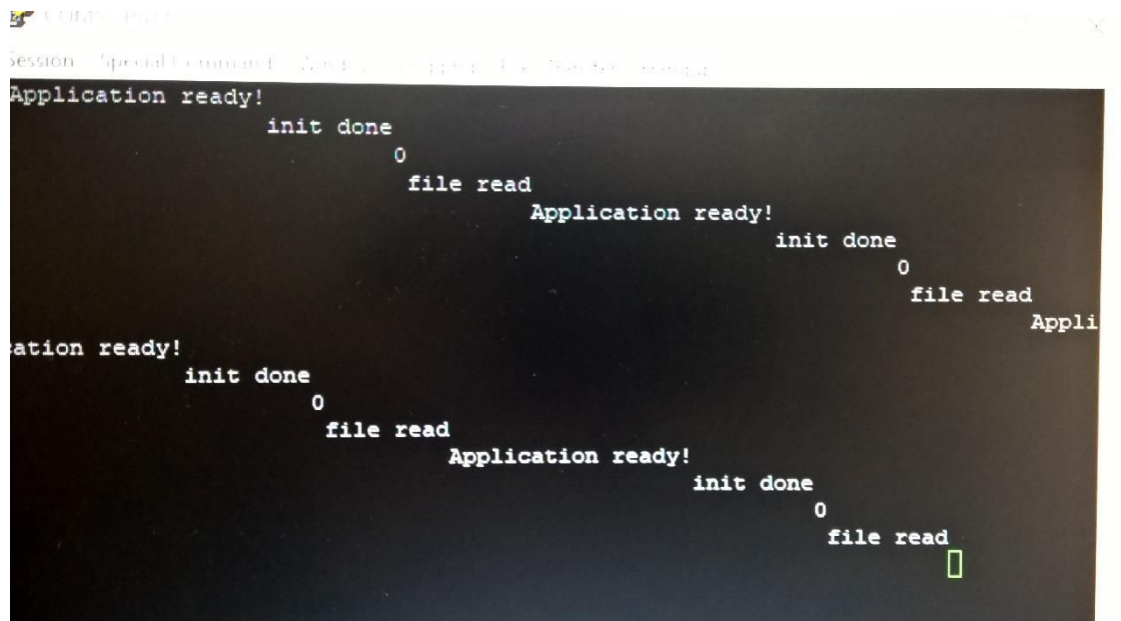
for(;;)

```
{
    osDelay(1);
}
```

/ USER CODE END 5 */*

```
}
```

Po uruchomieniu programu na konsoli pojawiły się komunikaty:



The screenshot shows a debugger window titled "Session - Special Command". The main area displays a call stack with the following text from top to bottom:

```
Application ready!  
    init done  
        0  
            file read  
                Application ready!  
                    init done  
                        0  
                            file read  
                                Appli  
ation ready!  
    init done  
        0  
            file read  
                Application ready!  
                    init done  
                        0  
                            file read
```

A green cursor is visible at the end of the last line of the call stack.

Następnie, po usunięciu breakpointów i pomocniczych funkcji printf, które powodowały opóźnienia i przycinanie się odtwarzanego pliku, piosenka odtwarzała się płynnie od początku do końca.

6. Dodanie biblioteki Helix do projektu.

Ten krok wymagał modyfikacji pliku *makefile*, które zostały przedstawione poniżej.

- dodanie plików biblioteki Helix o rozszerzeniu .c do *C_SOURCES* = \
- ```
C_SOURCES = \
...\
helix/mp3dec.c \
helix/mp3tabs.c \
helix/real/bitstream.c \
helix/real/buffers.c \
helix/real/dct32.c \
helix/real/dequant.c \
helix/real/dqchan.c \
helix/real/huffman.c \
helix/real/hufftabs.c \
helix/real/imdct.c \
helix/real/polyphase.c \
helix/real/scalfact.c \
helix/real/stproc.c \
helix/real/subband.c \
helix/real/trigtabs_fixpt.c
```
- dodanie podfolderów biblioteki Helix do *C\_INCLUDES* = \, gdzie -I to flaga Include:
- ```
C_INCLUDES = \  
...\  
-Ihelix/real \  
-Ihelix/pub
```
- pod *# compile gcc flags* zmodyfikowanie *CFLAGS* poprzez dodanie *-ffreestanding*

```
CFLAGS = $(MCU) $(C_DEFS) $(C_INCLUDES) $(OPT) -Wall -fdata-sections  
-ffunction-sections -ffreestanding
```

Po przeprowadzonych zmianach w pliku main można było dodać instrukcję *#include "mp3dec.h"* i rozpocząć korzystanie z nowo dodanej biblioteki.

7. Dekodowanie i odtworzenie pliku MP3.

W tej części należało doprowadzić do odtworzenia pliku MP3.

- **Charakterystyka plików MP3**

Plik MP3 to plik zawierający dźwięk zapisany w postaci cyfrowej z kompresją. Algorytm do kodowania i dekodowania tego typu sygnału pochodzi ze standardu MPEG, który dotyczy obróbki sygnałów audio. Jakością plików typu MP3 można manipulować zmieniając częstotliwość próbkowania, liczbę kodowanych kanałów oraz liczbę bitów odpowiadających pojedynczej próbce sygnałów audio. W dużym uproszczeniu, aby zapewnić małe rozmiary pliku wykorzystano fakt, że ludzkie ucho nie jest w stanie wychwycić niektórych dźwięków, można je więc pominąć.

Każdy plik standardu MPEG składa się z ramek. Pliki MP3 wspierają zmienny bitrate zależny od zawartości każdej ramki. Z każdą ramką powiązany jest więc nagłówek zawierający informacje do poprawnego dekodowania danych. Ważne jest aby zsynchronizować proces dekodowania z nadchodzącym strumieniem bitów.

- **Dekodowanie pliku**

Funkcja MP3Decode z biblioteki Helix posłużyła nam do dekodowania pliku:

```
short *outbuf;
read_ptr = Audio_Buffer;
HMP3Decoder hMP3Decoder;
static uint32_t AudioRemSize = 0;

int err = MP3Decode(hMP3Decoder, (unsigned char**)&read_ptr, &AudioRemSize,
outbuf, 0);
printf("%d", err);
printf("file decoded\n");
```

Ostatni argument w funkcji *MP3Decode* równy 0 oznacza, że dane są w zwyczajnym formacie MPEG (*normal MPEG format*). Wspomniana funkcja automatycznie aktualizuje wskaźnik do *read_ptr* oraz *outbuf*.

Po podłączeniu się poprzez Putty i uruchomieniu programu na konsoli wyświetlił się napis *file decoded*, a wartość *err* wynosiła 0, co oznaczało, że w *outbuf* pojawiła się ramka zdekodowanego pliku MP3. Odtworzone zostało również kilka dźwięków zawartych w ramce.

Niestety, pomimo starań nie udało nam się doprowadzić do odtworzenia pliku MP3.

8. Wnioski i podsumowanie.

- Niestety pomimo chęci, nie udało nam się zrealizować dodatkowej funkcjonalności zadania inżynierskiego. Było to związane głównie z brakiem czasu. Wiele zajęć laboratoryjnych spędziłyśmy przygotowując nasz projekt do działania w środowisku Eclipse, co okazało się dla nas problematyczne. W związku z koniecznością użycia płytki STM przy debuggowaniu kodu w następnych etapach zadania, nie byliśmy w stanie nadrobić straconego czasu poza zajęciami.
- Środowisko *STM32 CubeMX* okazało się prostym w użyciu i przydatnym narzędziem, dzięki któremu można było w prosty sposób wygenerować kod potrzebny do rozpoczęcia wykonywania zadania inżynierskiego. Dzięki temu programista nie znający się dogłębnie na tajnikach elektroniki jest w stanie stworzyć bazę swojego projektu.
- Użycie środowiska *GNU MCU Eclipse* okazało się bardzo problematyczne. Wiecznie przeskakujące okienka w trakcie debuggowania, wiele konfiguracji, mało intuicyjne ukrywanie przed użytkownikiem różnych opcji, błędy, które pomimo ich poprawienia nie znikają dopóki nie skompilowało się programu – wszystko to utrudniało pracę i doprowadziło do znacznych opóźnień. Narzędzia do debuggowania, chociaż wspierały nas przez większość czasu w sprawdzaniu postępów programu, to czasami ich zachowanie także nie pozwalało nam na szybkie zidentyfikowanie problemu.
- Pomimo niepowodzenia w implementacji funkcjonalności dodatkowej w postaci dekodowania MP3, z powodzeniem została zaimplementowana funkcjonalność podstawowego odtwarzacza nieskompresowanych plików dźwiękowych WAV. Dodatkowo, oprócz wyćwiczenia naszej cierpliwości do złośliwych środowisk nauczyłyśmy się w jaki sposób działać może odtwarzacz muzyki na płycie STM32.

9. Bibliografia.

- <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.557.4662&rep=rep1&type=pdf>
- <https://ep.com.pl/files/5156.pdf>
- Odtwarzacz plików typu wave na podstawie biblioteki *waveplayer* na płytce STM32F4 Discovery