Gadający STM32

Zastosowanie kodeka Speex do odtwarzania komunikatów głosowych

Konstruktorzy urządzeń elektronicznych często decydują się na dodanie do swoich projektów funkcji odtwarzania komunikatów głosowych. Na łamach Elektroniki Praktycznej prezentowane były już najróżniejsze „gadające" zegary, termometry czy też woltomierze. Najczęściej zadanie rejestracji, przechowywania i odtwarzania dźwięków powierzane było specjalizowanym układom „magnetofonów" półprzewodnikowych, czyli układom ISD14xx. Podyktowane to było stosunkowo prostą ich obsługą a przede wszystkim brakiem innej tak prostej i stosunkowo taniej technologii przechowywania dźwięku. Przechowywanie dźwięków w „surowej" postaci jest ogromnie nieefektywne pod względem wymaganej pojemności pamięci, natomiast dekompresja chociażby najpopularniejszego formatu dźwięku MP3 wymaga bardzo dużej, jak na typowe układy mikrokontrolerowe, mocy obliczeniowej, albo specjalizowanych i trudno dostępnych dekoderów sprzętowych.

Jeszcze kilka lat temu zastosowanie we własnym projekcie mikroprocesora 32-bitowego o wydajności kilkudziesięciu MIPS było zadaniem będącym poza zasięgiem przeciętnego amatora. Dziś w cenie zbliżonej do najwyższych modeli wiodącej prym rodziny mikrokontrolerów 8-bitowych można nabyć bardzo szeroką gamę wydajnych 32-bitowych mikrokontrolerów zbudowanych w oparciu o rdzeń ARM, posiadających szeroką gamę układów peryferyjnych. Niekwestionowanym hitem ostatnich miesięcy są mikrokontrolery STM32 wykorzystujące nowoczesny rdzeń Cortex-M3. Producent tych mikrokontrolerów, firma STMicroelectronics, przygotował aplikację (AN2812) demonstrującą możliwości kodeka Speex w zakresie nagrywania i odtwarzania komunikatów głosowych.

Czym jest kodek Speex?

Speex jest darmowym kodekiem o otwartym kodzie źródłowym, zaprojektowanym specjalnie dla przetwarzania mowy. Strona WWW projektu jest dostępna pod adresem *www.speex.org*. Kodek Speex został zaprojektowany z myślą o wykorzystaniu go w telefonii internetowej VoIP, jednak ze względu na stosunkowo niskie wymagania sprzętowe idealnie nadaje się do wykorzystania w systemach mikroprocesorowych, gdzie głównym kryterium doboru oprogramowania jest niskie zużycie pamięci programu oraz mocy obliczeniowej procesora.

Przykład zastosowania

W artykule zostanie przedstawiona aplikacja wykorzystująca kodek Speex do odtwarzania komunikatów głosowych przechowywanych w pamięci Flash mikrokontrolera STM32. Aplikacja została przygotowana dla kompilatora Raisonance, ale w prosty sposób może zostać dostosowana do innych kompilatorów dla mikrokontrolerów STM32. Omawiany w artykule program został przygotowany dla zestawu ZL27ARM (*www.kamami.pl*) z mikrokontrolerem STM32F103VBT6 posiadającym 128 kB pamięci Flash oraz 20 kB pamięci RAM, lecz może zostać uruchomiony praktycznie na dowolnej platformie sprzętowej z mikrokontrolerem STM32. Zestaw powinien być wyposażony w wyświetlacz LCD 2x16 znaków. Do zestawu należy dołączyć prosty filtr RC, pokazany na schemacie. Wyjście filtra należy podłączyć do wzmacniacza mocy sterującego głośnikiem. Możliwie jest podłączenie wyjścia PWM bezpośrednio z głośniczkiem znajdującym się na płytce zestawu ZL27ARM, jednakże głośność dźwięku będzie bardzo niska.

Przygotowanie nagrań dźwiękowych

Komunikaty głosowe, które mają być odtworzone przez przykładową aplikację należy nagrać na komputerze PC z wykorzystaniem dowolnego programu do rejestracji dźwięku, np. systemowego „Rejestratora Dźwięku" i zapisać je w formacie *.wav*. Ważne jest, aby częstotliwość próbkowania dźwięku podczas rejestracji była równa 8 kHz. Rozdzielczość próbkowania powinna wynosić 16-bitów. Następnie należy dokonać kompresji nagrań za pomocą programu *speexenc.exe* , który znajduje się w archiwum z przykładową aplikacją przygotowaną przez STMicroelectronics (AN2812). Program ten można również pobrać ze strony *www.speex.org/downloads*. Program *speexenc.exe* jest aplikacją pracującą w wierszu poleceń. W celu otrzymania pliku zakodowanego kodekiem Speex, który będzie można odtworzyć za pomocą aplikacji demonstracyjnej należy wywołać program *speexenc.exe* z następującymi parametrami :

```
speexenc.exe -n --quality 4 input_file output_file
```

Parametr `input_file` określa ścieżkę dostępu do pliku z nagranym komunikatem dźwiękowym, natomiast parametr `output_file` określa ścieżkę dostępu do pliku wyjściowego.

Aby możliwe było wykorzystanie tak otrzymanych danych w kodzie źródłowym, należy dokonać konwersji pliku zawierającego dane binarne do pliku tekstowego zawierającego tablicę danych zawierającą liczbową reprezentację każdego bajtu pliku binarnego. Do tego celu można wykorzystać przykładowo program Hex Workshop i za pomocą opcji Export wygenerować plik źródłowy języka C z tablicą zawierającą dane z otwartego pliku. W omawianej aplikacji, dane reprezentujące nagrane komunikaty znajdują się w pliku *voice.h*. Oprócz tego, w pliku voice.h została zdefiniowana struktura `Sound_t`. Definicja struktury jest przedstawiona następująca:

```
typedef struct
{
  u8 * Pointer;
  u16 Length;
}Sounds_t;
```

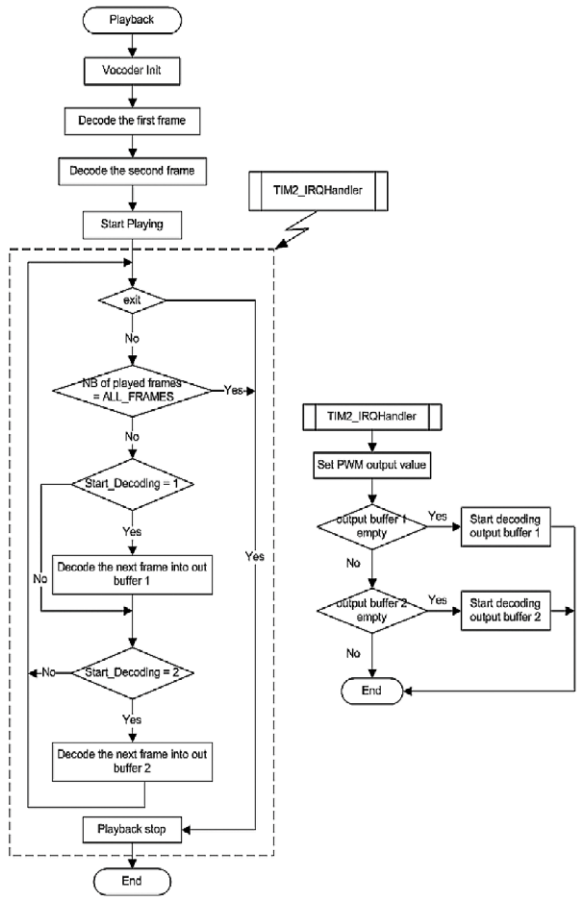
Struktura ta składa się z dwóch pól: pierwsze pole struktury jest wskaźnikiem do typu u8 i będzie przechowywać adres początku tablicy z zakodowanymi danymi, natomiast drugie pole jest typu u16 i będzie przechowywać długość komunikatu w ramkach. Długość komunikatu w ramkach można obliczyć poprzez podzielenie rozmiaru pliku **.spx* (czyli rozmiaru tablicy) przez liczbę bajtów w ramce, czyli przez liczbę 20. Oprócz struktury w pliku *voice.h* została zdefiniowana również tablica typu `Sound_t` przechowująca dane wszystkich zapisanych w pliku komunikatów dźwiękowych. Kod inicjujący tablicę przedstawiono niżej:

```
Sounds_t Sounds[3]={
{rawData1, 311},
{rawData2, 96},
{rawData3, 71},
};
```

W celu uzyskania danych określających konkretny komunikat wystarczy odwołać się do wybranej pozycji tablicy, np.: `Sounds[0].Pointer` zwróci adres komunikatu, a `Sounds[0].Length` zwróci jego długość. Dzięki temu można w prosty sposób odtworzyć wszystkie komunikaty za pomocą pętli iterującej po wszystkich pozycjach tablicy, co zostanie wykorzystane w programie demonstracyjnym.

Dekodowanie dźwięku z pamięci Flash

Ogólny algorytm odtwarzania komunikatów dźwiękowych przedstawiono na **rys. 1**. Aplikacja wykorzystuje dwa bufory do dekodowania i odtwarzania nagrania. Dekodowanie dźwięku zapisanego w pamięci Flash mikrokontrolera dokonywane jest w ramach funkcji *PlaySound*. Funkcja ta przyjmuje dwa parametry: wskaźnik do obszaru pamięci, w którym znajduje się zakodowany kodekiem Speex dźwięk oraz jego długość w ramkach. Każda ramka zakodowanego nagrania składa się z dwudziestu bajtów, natomiast ramka zdekodowanego nagrania zajmuje 160 bajtów. Wynika z tego, iż współczynnik kompresji nagrania wynosi 1:8.



Rys. 1.

Kod funkcji *PlaySound* przedstawiono na **list. 1**.

Działanie funkcji rozpoczyna się od zdekodowania dwóch pierwszych ramek nagrania i wypełnienia obydwu buforów wyjściowych zdekodowanymi danymi. Dekodowanie każdej ramki nagrania przebiega w następujący sposób :

- tablica `input_bytes` musi zostać wypełniona odczytaną z pamięci Flash ramką sygnału,
- następnie zawartość tablicy `input_bytes` jest kopiowana, za pomocą funkcji *speex_bits_read_from*, do struktury `bits`, stanowiącej „wejście" dekodera Speex,
- ramka znajdująca się w strukturze `bits` jest dekodowana do bufora wyjściowego `OutBuffer`, za pomocą funkcji *speex_decode_int*.

Po zdekodowaniu dwóch ramek i wypełnieniu obydwu buforów wyjściowych do zmiennej `Play` zapisywana jest liczba 1. Zmienna `Play` sygnalizuje fakt wypełnienia buforów zdekodowanymi danymi i zezwala na ich odtworzenie przez procedurę obsługi przerwania od timera TIM2. Pozostała część nagrania jest dekodowana w pętli, aż do osiągnięcia końca nagrania, czyli do momentu, aż zmienna `NB_Frames` zrówna się ze zmienną `Length`. Stan zmiennej `Start_Decoding` określa, który z buforów został opróżniony i powinien zostać wypełniony nowymi danymi. Stan tej zmiennej jest modyfikowany przez procedurę odtwarzania dźwięku z buforów wyjściowych.

List. 1.

```
void PlaySound(const u8 *Sound, u16 Lenght)
{
    vu16 i;
    for(i=0;i<ENCODED_FRAME_SIZE; i++)
      input_bytes[i] = *(Sound + sample_index++);

    speex_bits_read_from(&bits, input_bytes, ENCODED_FRAME_SIZE);
    speex_decode_int(dec_state, &bits, (spx_int16_t*)OUT_Buffer[0]);

    for(i=0;i<ENCODED_FRAME_SIZE; i++)
      input_bytes[i] = *(Sound + sample_index++);

    speex_bits_read_from(&bits, input_bytes, ENCODED_FRAME_SIZE);
    speex_decode_int(dec_state, &bits, (spx_int16_t*)OUT_Buffer[1]);

    NB_Frames++;

    Play = 1;

    while(NB_Frames < Lenght)
    {
      if(Start_Decoding == 1)
      {
        for(i=0;i<ENCODED_FRAME_SIZE; i++)
          input_bytes[i] = *(Sound + sample_index++);

        speex_bits_read_from(&bits, input_bytes, ENCODED_FRAME_SIZE);
        speex_decode_int(dec_state, &bits, (spx_int16_t*)OUT_Buffer[0]);

        Start_Decoding = 0;
        NB_Frames++;
      }

      if(Start_Decoding == 2)
      {
        for(i=0;i<ENCODED_FRAME_SIZE; i++)
          input_bytes[i] = *(Sound + sample_index++);

        speex_bits_read_from(&bits, input_bytes, ENCODED_FRAME_SIZE);
        speex_decode_int(dec_state, &bits, (spx_int16_t*)OUT_Buffer[1]);

        Start_Decoding = 0;
        NB_Frames++;
      }
    }
    Play = 0;
    sample_index = 0;
    NB_Frames = 0;
    outBuffer = OUT_Buffer[0];
}
```

Odtwarzanie dźwięku z buforów wyjściowych

Jako przetwornik cyfrowo-analogowy wykorzystywany jest timer 1 pracujący w trybie PWM. Współczynnik wypełnienia sygnału generowanego przez timer 1 jest aktualizowany z częstotliwością 8 kHz w ramach obsługi przerwania od timera 2. Kod handlera przerwania od timera TIM2 przedstawiono na **list. 2**.

Na początku procedury następuje załadowanie do rejestru ARR wartości, od której licznik rozpoczyna odliczanie oraz wyzerowanie flagi przerwania. Następnie sprawdzany jest stan zmiennej `Play`, która określa czy dane z bufora wyjściowego mają być przekazane na wyjście. Jeżeli wartość zmiennej jest różna od zera, nastąpi załadowanie do rejestru CCR1 licznika TIM1 wartości określającej aktualny współczynnik wypełnienia generowanego sygnału PWM oraz sprawdzenie, czy osiągnięto koniec któregokolwiek z buforów wyjściowych. W przypadku, gdy cały aktualnie odtwarzany bufor został odczytany, następuje przełączenie odtwarzania na drugi z buforów. Jeśli natomiast bufor nie został w pełni odtworzony, inkrementowany jest wskaźnik `outBuffer`, który wskazuje na aktualną pozycję w buforze wyjściowym. W przypadku, gdy zmienna `Play` jest wyzerowana, do rejestru CCR1 timera TIM1 zapisywana jest stała wartość 0x200, odpowiadająca w przybliżeniu wartości „masy analogowej" na wyjściu sygnału, czyli połowie napięcia maksymalnego generowanego przez układ PWM timera TIM1, gdyż w przypadku omawianej aplikacji timery TIM1 i TIM2 pracują nieprzerwanie przez cały czas działania aplikacji. Oryginalna aplikacja opracowana przez firmę STMicroeletronics działa nieco inaczej, gdyż timery są aktywowane tylko na czas odtwarzania komunikatu, co objawia się niestety nieprzyjemnymi trzaskami w momencie włączania i wyłączania generatora PWM. W przedstawianej w artykule aplikacji ten problem nie występuje, kosztem niewielkiego obciążenia procesora spowodowanego ciągłym występowaniem przerwania od timera TIM2.

Inicjalizacja układów peryferyjnych

Ponieważ aplikacja wykorzystuje podczas pracy dwa układy timerów konieczne jest ich właściwe zainicjalizowanie. Inicjalizacja wykorzystywanych układów peryferyjnych realizowana jest za pomocą funkcji *Vocoder_Init*, której kod przedstawiono na **list. 3**.

Timer TIM1 jest skonfigurowany do pracy w trybie PWM. Wyjście PA8 jest skonfigurowane jako wyjście funkcji alternatywnej, czyli w tym przypadku wyjście PWM timera TIM1. Timer TIM2 został skonfigurowany do generowania przerwania z częstotliwością 8 kHz.

Konfiguracja układu przerwań

Ponieważ przesyłanie danych z buforów wyjściowych na wyjście analogowe odbywa się w ramach obsługi przerwania od timera TIM2, konieczne jest odpowiednie skonfigurowanie układu przerwań. Kod funkcji odpowiedzialnej za konfigurację układu przerwań przedstawiono na **list. 4**.

Główny kod programu

Zasadnicza część aplikacji odtwarzającej komunikaty z pamięci Flash mikrokontrolera jest stosunkowo prosta, gdyż jej celem jest jedynie praktyczne przedstawienie wykorzystania kodeka Speex do odtwarzania uprzednio przygotowanych komunikatów dźwiękowych. Kod głównej funkcji programu przedstawiony jest na **list. 5**.

List. 2.
```
void TIM2_IRQHandler(void)
{
  TIM2->ARR = TIM2ARRValue;
  TIM2->SR = TIM_INT_Update;

  if(Play)
  {
    TIM1->CCR1 = ((*outBuffer>>6)) + 0x200 ;

    if(outBuffer == &OUT_Buffer[1][159])
    {
      outBuffer = OUT_Buffer[0];
      Start_Decoding = 2;
    }
    else if(outBuffer == &OUT_Buffer[0][159])
    {
      outBuffer = OUT_Buffer[1];
      Start_Decoding = 1;
    }
    else
    {
      outBuffer++;
    }
  }
  else
  {
  TIM1->CCR1 = 0x200 ;
  }
}
```

List. 3.
```
void Vocoder_Init(void)
{
  /* Peripherals InitStructure define ----------------------------------------
-*/
  GPIO_InitTypeDef GPIO_InitStructure;
  ADC_InitTypeDef ADC_InitStructure;
  TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
  TIM_OCInitTypeDef  TIM_OCInitStructure;

  /* TIM1 configuration ------------------------------------------------------
-*/
  TIM_DeInit(TIM1);
  TIM_OCStructInit(&TIM_OCInitStructure);
  TIM_TimeBaseStructInit(&TIM_TimeBaseStructure);
  GPIO_StructInit(&GPIO_InitStructure);

  /* Configure PA.08 as alternate function (TIM1_OC1) */
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOA, &GPIO_InitStructure);
  GPIO_ResetBits(GPIOA, GPIO_Pin_8);

  /* TIM1 used for PWM genration */
  TIM_TimeBaseStructure.TIM_Prescaler = 0x00; /* TIM1CLK = 72 MHz */
  TIM_TimeBaseStructure.TIM_Period = 0x3FF; /* 10 bits resolution */
  TIM_TimeBaseStructure.TIM_ClockDivision = 0;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;
  TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);

  /* TIM1's Channel1 in PWM1 mode */
  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
  TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
  TIM_OCInitStructure.TIM_OutputNState = TIM_OutputNState_Enable;
  TIM_OCInitStructure.TIM_Pulse = 0x200;/* Duty cycle: 50%*/
  TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
  TIM_OCInitStructure.TIM_OCNPolarity = TIM_OCNPolarity_High;
  TIM_OCInitStructure.TIM_OCIdleState = TIM_OCIdleState_Set;
  TIM_OCInitStructure.TIM_OCNIdleState = TIM_OCIdleState_Reset;
  TIM_OC1Init(TIM1, &TIM_OCInitStructure);

  TIM_OC1PreloadConfig(TIM1, TIM_OCPreload_Enable);
  TIM_ARRPreloadConfig(TIM1, ENABLE);

  /* TIM2 configuration ------------------------------------------------------
-*/
  TIM_DeInit(TIM2);
  TIM_OCStructInit(&TIM_OCInitStructure);
  TIM_TimeBaseStructInit(&TIM_TimeBaseStructure);
  /* TIM2 used for timing, the timing period depends on the sample rate */
  TIM_TimeBaseStructure.TIM_Prescaler = 0x00;    /* TIM2CLK = 72 MHz */
  TIM_TimeBaseStructure.TIM_Period = TIM2ARRValue;
  TIM_TimeBaseStructure.TIM_ClockDivision = 0x0;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);

  /* Output Compare Inactive Mode configuration: Channel1 */
  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Inactive;
  TIM_OCInitStructure.TIM_Pulse = 0x0;
  TIM_OC1Init(TIM2, &TIM_OCInitStructure);
  TIM_OC1PreloadConfig(TIM2, TIM_OCPreload_Disable);
}
```

List. 4.
```
void InterruptConfig(void)
{
  NVIC_InitTypeDef NVIC_InitStructure;
  NVIC_DeInit();
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x00);

  NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQChannel;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
```

List. 5.
```
int main(void)
{
  vu8 i;
  vu32 j;
  Demo_Init();                                          //
  LCD_Initialize();                                     //
  Speex_Init();                                         // Inicjalizacja
  Vocoder_Init();                                       //
  Vocoder_Start();                                      //

  LCD_GoTo(0,0);                                        //
  LCD_WriteText(„Speex Demo - EP”);                     //
  LCD_GoTo(0,1);                                        // Ekran powitalny
  LCD_WriteText(„www.ep.com.pl”);                       //

for(j = 0; j < 0x3FFFFF; j++);                          // Opóźnienie
  while(1)                                              // pętla nieskończona
  {
  for(i = 0; i < 9; i++)
    {
    LCD_GoTo(0,1);                                      //
    LCD_WriteText(„Komunikat nr „);                     // Wyświetlenie napisu
    LCD_WriteData(i+’0’);                               //

    PlaySound(Sounds[i].Pointer, Sounds[i].Length); // Odtworzenie dźwięku

    for(j = 0; j < 0xFFFFF; j++);                       // Opóźnienie
    }
  }
}
```

Działanie funkcji rozpoczyna się od inicjalizacji wykorzystywanych układów peryferyjnych oraz kodeka. Następnie na wyświetlaczu LCD wyświetlany jest ekran powitalny i po krótkiej chwili oczekiwania następuje odtworzenie komunikatów w pętli nieskończonej. Wykorzystana do tego celu zostaje tablica `Sounds` przechowująca adresy oraz długości poszczególnych komunikatów. Oprócz odtworzenia komunikatu na wyświetlaczu LCD jest wyświetlany numer aktualnie odtwarzanego komunikatu.

Podsumowanie

Przedstawiona w artykule aplikacja demonstracyjna zajmuje, łącznie z nagranymi komunikatami, około 30 kB pamięci Flash mikrokontrolera. Do dyspozycji programisty pozostaje jeszcze spora ilość miejsca, w zależności od zastosowanego typu mikrokontrolera, na zasadniczą aplikację, wykorzystującą kodek Speex do odtwarzania dźwięku. Trudno bowiem sobie wyobrazić, iż odtwarzanie komunikatów dźwiękowych może być jedyną funkcją, do której zaprzęgnięty zostanie mikrokontroler z rodziny STM32, stanowi to rzecz jasna tylko dodatek do szerokiego grona aplikacji, w których zastosowanie mogą znaleźć rewelacyjne mikrokontrolery STM32.

Radosław Kwiecień, EP
radoslaw.kwiecien@ep.com.pl