Zeszyty Naukowe Wydziału Elektrotechniki i Automatyki Politechniki Gdańskiej Nr 51

XXVI Seminarium

ZASTOSOWANIE KOMPUTERÓW W NAUCE I TECHNICE 2016

Oddział Gdański PTETiS

DSPElib - BIBLIOTEKA C++ DO SZYBKIEJ IMPLEMENTACJI WIELOSZYBKOŚCIOWYCH ALGORYTMÓW PRZETWARZANIA SYGNAŁÓW

Marek BLOK

Politechnika Gdańska, Wydział Elektroniki, Telekomunikacji i Informatyki, Katedra Sieci Teleinformacyjnych, tel.: 58 347 2779, e-mail: mblok@eti.pg.gda.pl

Streszczenie: W pracy przedstawiono opracowaną bibliotekę C++, DSPElib – Digital Signal Processing Engine library, pozwalającą na prostą i szybką implementację wieloszybkościowych algorytmów przetwarzania sygnałów zawierających sprzężenia zwrotne, a co za tym idzie na szybkie prototypowanie tego typu algorytmów i włączanie ich do autonomicznych aplikacji przeznaczonych na platformę Windows lub Linux.

Słowa kluczowe: cyfrowe przetwarzanie sygnałów, wieloszybkościowe przetwarzanie sygnałów, biblioteka C++, szybkie prototypowanie algorytmów.

1. WPROWADZENIE

W cyfrowym przetwarzaniu sygnałów (DSP – Digital Signal Processing) szczególnym wyzwaniem jest wykorzystywanie algorytmów zawierających sprzężenie zwrotne, które sprawia, że kolejne wartości próbek wyjściowych algorytmu zależą od wartości poprzednich jego próbek wyjściowych. Przykładem takich algorytmów są cyfrowe filtry typu IIR (Infinite Impulse Response) [1] albo algorytmy adaptacyjne, których przykładem mogą być algorytmy synchronizacji symbolowej [2]. Wyklucza to blokową implementację, w której kolejne etapy przetwarzania można realizować odrębnie, niezależnie przetwarzając większe segmenty sygnału. Skomplikowanie implementacji w efekcie występowania sprzężenia zwrotnego jest szczególnie istotne na etapie prototypowania algorytmu, gdy docelowa struktura i parametry algorytmu przetwarzania sygnałów nie są jeszcze ściśle określone i są zmieniane w rezultacie testowania kolejnych wersji kodu, co skutkuje częstym wprowadzaniem poprawek. Nawet drobna zmiana koncepcji przetwarzania może w takim przypadku skutkować znacznymi zmianami w kodzie. Jeszcze większy problem stanowi implementacja algorytmów wieloszybkościowych (multirate), w których różne fragmenty algorytmu pracują z różnymi szybkościami próbkowania [3, 4].

W przypadku projektowania i implementacji algorytmów DSP pierwszym etapem typowo jest przetworzenie koncepcji do postaci schematu blokowego, po czym następuje wstępna implementacja i testowanie, zwykle w środowisku MATLAB [5] (bądź OCTAVE [6]), a końcowym etapem jest implementacja i testowanie docelowej aplikacji przygotowanej w języku C/C++. Dodatkowo w efekcie testów często trzeba powrócić do wcześniejszych etapów projektowania lub implementacji.

Zaletą środowiska typu MATLAB jest prostota implementacji i dostępność znacznego zaplecza algorytmów przetwarzania oraz łatwość wglądu w sygnały z dowolnego punktu algorytmu. Środowisko to interpretuje polecenia zapisane w postaci skryptów, co ogranicza jego efektywność. W przypadku implementacji blokowych nie jest istotnym ograniczeniem ze względu na optymalizację tego środowiska pod kątem przetwarzania macierzowego. Jednak skrypty zawierające pętle sprzężenia zwrotnego, w których przetwarzanie musi być realizowane potokowo, próbka po próbce, są przetwarzanie bardzo wolno.

Problemy te były motywacją do opracowania biblioteki C++ umożliwiającej implementację algorytmów DSP w ścisłym powiązaniu ze schematem blokowym, z pominięciem etapu jego realizacji w MATLABie i ograniczenie się do implementacji i testowania wybranych elementów projektowanego algorytmu. Ścisłe powiązanie implementacji w C++ ze schematem blokowym pozwala na łatwe przenoszenie modyfikacji schematu blokowego do kodu źródłowego aplikacji, a co za tym idzie na łatwe i szybkie testowanie różnych wariantów powstających na etapie prototypowania algorytmu. Jednocześnie z uzyskaniem większej wydajności przetwarzania dla realizowanych algorytmów wiąże się możliwość pominięcia czasochłonnej niskopoziomowej ich implementacji w C/C++.

Prezentowana biblioteka jest wykorzystywana przez jej autora w pracy badawczej oraz dydaktyce. Z jej użyciem powstały aplikacje realizujące zaawansowane algorytmy przetwarzania sygnałów pracujące w czasie rzeczywistym na komputerach klasy PC, m.in. cyfrowy demodulator FSK [7] i edukacyjny model modulatora/demodulatora OFDM [8]. Studenci, posługując się biblioteką, w ramach zajęć projektowych oraz laboratorium z przedmiotu „Zaawansowane przetwarzanie sygnałów telekomunikacji cyfrowej", już po krótkim przygotowaniu mogą samodzielnie implementować, modyfikować i testować autonomiczne aplikacje DSP korzystając z ogólnie dostępnych narzędzi. W ramach tych zajęć biblioteka wykorzystywana jest w realizacji takich zadań jak: polifazowa implementacja filtrów interpolacyjnych i decymacyjnych, projektowanie i implementacja interpolowanych filtrów FIR (I-FIR) [9] czy implementacja i badanie banku filtrów analizujących i syntezujących [10]. Przy użyciu biblioteki wytworzono również narzędzia wykorzystywane w ramach zajęć laboratoryjnych: PiAPS_sound – pozwalające na badanie właściwości przetworników A/C i C/A kart dźwiękowych [11], a także TELEsound – oprogramowanie wytworzone na potrzeby laboratorium z przedmiotu „Podstawy telekomunikacji" [12].

2. KONCEPCJA I KOMPONENTY BIBLIOTEKI

Zalążek koncepcji biblioteki DSPElib stanowiła aplikacja miCePS rozwijana w latach 1993-2001, która była pomyślana jako uzupełnienie oprogramowania DSPS [13], realizującego przetwarzanie blokowe, o możliwość realizacji przetwarzania potokowego z pętlą sprzężenia zwrotnego. Pozwalała ona na ograniczenie bardzo czasochłonnego, niezbędnego na etapie rozwoju algorytmów, testowania i modyfikowania "czystej" implementacji C++. W związku z ograniczeniami tej aplikacji, takimi jak: implementacja w formie interpretera skryptów, brak możliwości wykorzystania algorytmów wieloszybkościowych oraz ograniczone możliwości diagnostyki oprogramowywanych algorytmów, rozwój tej aplikacji został zarzucony.

Doświadczenia nabyte w trakcie rozwoju aplikacji miCePS stanowiły punkt wyjścia w opracowaniu prezentowanej w tym artykule biblioteki DSPElib, której pierwsza wersja powstała w 2005 roku. Modyfikacje założeń były jednak na tyle istotne, że biblioteka została opracowana całkowicie od nowa przy następujących założeniach:

- zapewnienie wsparcia dla implementacji algorytmów zawierających sprzężenia zwrotne oraz rozwiązania wieloszybkościowe,
- umożliwienie wytworzenia w C++ samodzielnej aplikacji realizującej algorytm pracujący w czasie rzeczywistym,
- zapewnienie prostej aktualizacji zastosowanych rozwiązań, tak żeby aktualizacja jądra biblioteki nie wymagała modyfikacji kodu właściwej aplikacji,
- prostota implementacji algorytmów DSP w sposób bezpośrednio powiązany ze schematem blokowym algorytmu, proste definiowanie połączeń oraz relacji czasowych, otwartość biblioteki,
- prostota diagnozowania algorytmów DSP.

Podczas tworzenia biblioteki szczególną uwagę zwrócono na wykorzystanie narzędzi ogólnie dostępnych, udostępnianych na zasadach licencji wolnego oprogramowania. Stąd biblioteka ta jest przystosowana do kompilacji z użyciem kompilatora g++ dostępnego zarówno na platformie Windows, jak i Linux.

2.1. Struktura biblioteki

W opracowanej bibliotece można wyróżnić obsługę bloczków przetwarzania oraz obsługę zegarów. Każdy bloczek przetwarzania jest reprezentowany przez odrębną klasę C++, tak aby zapewnić łatwe przenoszenie schematu algorytmu DSP do kodu C++. Wszystkie bloczki przetwarzania obsługiwane przez bibliotekę są implementowane jako klasy pochodne od klas DSP_block lub DSP_source.

Klasa bazowa DSP_block implementuje podstawowe mechanizmy obsługi bloków, których próbki wyjściowe są generowane w oparciu próbki wejściowe. Szczególnym przypadkiem tego typu bloczków są ujścia sygnału, które przetwarzają próbki wejściowe bez generowania próbek wyjściowych, np. bloczek zapisu sygnału do pliku (DSPu_FILEoutput). Z kolei klasa DSP_source zapewnia podstawowe mechanizmy obsługi bloków stanowiących źródła sygnałów, czyli takich, które generują próbki wyjściowe, np. generator szumu (DSPu_rand) albo źródło podające na wyjście kolejne próbki pobierane z pliku (DSPu_FILEinput). W przypadku źródeł sterowanych, które generują próbki wyjściowe na podstawie parametrów otrzymywanych na ich wejściach, klasa implementująca je jest tworzona jako klasa pochodna obydwu klas bazowych: DSP_source i DSP_block. Przykładem może być bloczek DSPu_DCO realizujący generator, na którego wejście podaje się sygnały korekty jego częstotliwości i fazy.

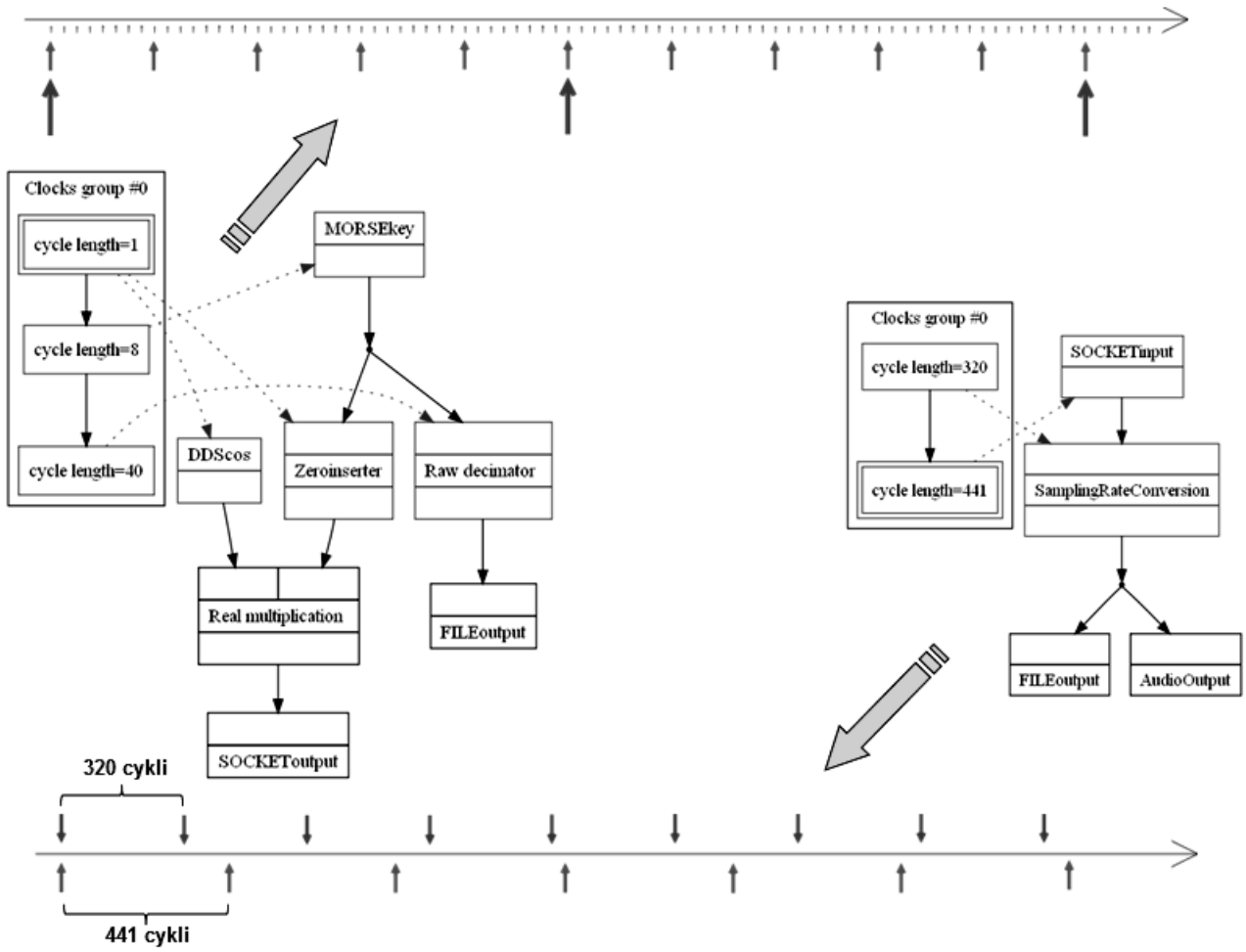
Ponieważ biblioteki ma służyć do implementacji algorytmów wieloszybkościowych, to bardzo ważna jest obsługa zegarów taktujących pracę poszczególnych segmentów algorytmu. Rdzeniem biblioteki jest klasa DSP_clock implementująca silnik przetwarzania kontrolujący wszystkie zdefiniowane w algorytmie zegary i inicjujący działanie powiązanych z nimi źródeł. Źródła te generując próbki wyjściowe pobudzają zarazem do działania pozostałe bloczki przetwarzania.

Silnik biblioteki umożliwia definiowanie zarówno zegarów działających synchronicznie, jak i asynchronicznie. Zegary synchroniczne pracują z szybkością stanowiącą L/M-krotność szybkości zegara referencyjnego, gdzie L i M są dodatnimi liczbami całkowitymi. Z kolei zegary asynchroniczne są aktywowane sygnałowo, np. bloczek DSPu_GardnerSampling stosowany w demodulatorze cyfrowym aktywuje zegar wyjściowy i generuje próbkę wyjściową, gdy na podstawie próbek wejściowych podjęto decyzję, że bieżąca próbka wejściowa jest interesującą nas próbką reprezentującą symbol nadany przez modulator cyfrowy [2].

2.2. Jądro biblioteki

Obsługa zegarów w jądrze biblioteki jest realizowana na zasadach symulacji zdarzeniowej, gdzie zdarzenia powiązane są z początkami kolejnych cykli poszczególnych zegarów. Przykładowe algorytmy wieloszybkościowe pokazuje rys. 1. Po lewej stronie pokazano algorytm generujący sygnał kluczowany zgodnie z kodem Morse'a. Sygnał kluczujący jest generowany w bloczku DSPu_MORSEkey z szybkością 1000 Sa/s. Sygnał ten jest zapisywany do pliku (DSPu_FILEoutput) po 5-krotnej decymacji (szybkość 200 Sa/s). Jednocześnie po 8-krotnej interpolacji moduluje on sinusoidalny sygnał nośny generowany w bloczku DSPu_DDScos (8000 Sa/s). Tak zmodulowany sygnał jest przekazywany przez sieć TCP/IP do kolejnej aplikacji z użyciem bloczka DSPu_SOCKEToutput. Występującym w tym algorytmie szybkościom próbkowania odpowiadają następujące długości cykli zegarów: 1, 5 i 40. Jeden cykl zegara odpowiada tu długości cyklu podstawowego określonej jako najmniejszy wspólny podzielnik odstępów próbkowania wszystkich zegarów występujących w algorytmie. Występowanie na osi czasu początków cykli poszczególnych zegarów zaznaczono strzałeczkami na górze rysunku 1. Zauważmy, że w tym przypadku zdarzenia obsługi zegarów można powiązać z obsługą zdarzeń najszybszego zegara, którego cykl odpowiada cyklowi podstawowemu realizowanego algorytmu. Wolniejsze zegary są po prostu obsługiwane w wybranych cyklach tego zegara.

Nieco inna sytuacja występuje w przypadku dwuszybkościowego algorytmu pokazanego po prawej stronie rys. 1. Algorytm ten odbiera poprzez sieć IP (DSPu_SOCKETinput) sygnał generowany przez poprzednio omawiany algorytm i odtwarza go z użyciem karty dźwiękowej (DSPu_AudioOutput) z szybkością 11025 Sa/s. Ponieważ odbierany sygnał cechuje szybkość próbkowania 8000 Sa/s, to konieczna jest realizacja zmiany szybkości próbkowania (DSPu_SampleRateConversion) z krotnością 441/320. Stąd w tym algorytmie cykle zegarów



Rys. 1. Przykładowe algorytmy wieloszybkościowe wraz ze wskazanymi na osiach czasu początkami cykli poszczególnych zegarów

wejściowego i wyjściowego wynoszą odpowiednio 441 i 320 cykli podstawowych. Jak widać na osi czasu przedstawionej w dolnej części rysunku 1 obydwa zegary są obsługiwane w odmiennych chwilach czasu, a ponadto poszczególne zdarzenia (początki cykli zegarów) są od siebie oddzielone o około kilkaset cykli podstawowych, co uzasadnia stosowanie symulacji zdarzeniowej.

Jądro biblioteki w pierwszym kroku określa najbliższą chwilę, w której aktywowany jest przynajmniej jeden zegar. Dla tej chwili określane są wszystkie aktywowane zegary, których kolejny cykl zaczyna się w danej chwili. Dla znalezionych zegarów są tworzone dwie listy źródeł sygnałów (bloków pochodnych od klasy DSP_source) powiązanych z aktywowanymi zegarami. Pierwsza, to lista źródeł, do których należy przekazać informację o rozpoczęciu się cyklu zegara, w którym dane źródło będzie aktywowane. Powiadomienia te są istotne przede wszystkim dla asynchronicznych źródeł sterowanych sygnałowo. Informują one źródło o tym, że w danym cyklu pojawi się na jego wejściu próbka sterująca jego pracą. Po przekazaniu powiadomień wywoływane są procedury obsługi wyjść źródeł znajdujących się na drugiej liście zawierającej źródła generujące w danym cyklu próbki wyjściowe.

W ramach obsługi źródła sygnału obliczone próbki wyjściowe są przekazywane do podłączonych do niego bloczków, które z kolei przekazują te próbki do kolejnych bloczków. W szczególnych przypadkach obsługa źródła może być odroczona, gdy źródło oczekuje na dane zewnętrzne. Przykładowo źródło DSPu_AudioInput może nadal oczekiwać na próbki pozyskiwane za pomocą karty dźwiękowej a źródło DSPu_SocketInput na próbki przesyłane przez sieć. W takim przypadku po przetworzeniu pozostałych źródeł procedura obsługi zegarów ponownie podejmie próbę wykonania procedur obsługi tych źródeł.

Dodatkowo, w ramach obsługi źródeł, mogą być aktywowane zegary asynchroniczne sterowane sygnałowo. Trafiają one do zbioru zegarów aktywowanych w danym cyklu i zostaną obsłużone po zakończeniu obsługi wszystkich bieżących źródeł. Sytuacja, gdy główna procedura obsługi zegarów stwierdzi, że w bieżącym cyklu obsługi nie udało się przetworzyć żadnego źródła i jednocześnie żadne źródło nie oczekuje na dane zewnętrzne oraz lista źródeł aktywowanych sygnałowo jest pusta oznacza, że w działaniu algorytmu wystąpił błąd, najprawdopodobniej związany z nieprawidłowo zdefiniowanymi relacjami pomiędzy zegarami i biblioteka raportuje problem.

3. UŻYTKOWANIE BIBLIOTEKI

Korzystając z prezentowanej biblioteki programista może oprogramować algorytm przetwarzania sygnałów bezpośrednio w oparciu o jego schemat blokowy. Typowe bloki przetwarzania deklaruje się jako obiekty, a połączenia pomiędzy blokami definiuje się jednym poleceniem (DSP_connect) wskazując obiekty reprezentujące łączone bloki oraz nazwy odpowiednich wejść i wyjść. Użytkownik biblioteki nie musi wdawać się w szczegóły implementacji bloczków oraz relacji pomiędzy nimi, co jest znacznym ułatwieniem zwłaszcza, gdy w algorytmie występują sprzężenia zwrotne, a bloczki pracują z różnymi szybkościami próbkowania.

Poniżej przedstawiono kod źródłowy aplikacji implementującej algorytm, którego schemat pokazano po prawej stronie rys. 1.

```
void main(void)
{
  DSP_clock_ptr MasterClock;
  MasterClock = DSP_clock::CreateMasterClock();

  DSP_LoadCoef coef_info; DSP_float h_SRC[9261];
  coef_info.Open("LPF_8000_11025.coef", "matlab");
  coef_info.Load(h_SRC, 9261);

  DSPu_SOCKETinput in_socket(MasterClock,
                             "127.0.0.1", true, 0x00000003);
  DSPu_SamplingRateConversion SRC(MasterClock, 441, 320,
                             9261, h_SRC);
  DSPu_AudioOutput AudioOut(11025);
  DSPu_FILEoutput WAVEfile("morse.wav",
        DSP_ST_short, 1, DSP_FT_wav, 11025);

  DSP_connect(in_socket.Output("out"), SRC.Input("in"));
  DSP_connect(SRC.Output("out"), AudioOut.Input("in"));
  DSP_connect(SRC.Output("out"), WAVEfile.Input("in"));

  DSP_component::CheckInputsOfAllComponents();
  DSP_clock::SchemeToDOTfile(MasterClock, "schemat.dot");

  do {
    DSP_clock::Execute(MasterClock, 2000);
  } while (in_socket.GetBytesRead() != 0);

  DSP_clock::FreeClocks();
}
```

Kolejne segmenty programu realizują następujące zadania:

- zdefiniowanie głównego zegara,
- wczytanie współczynników filtru interpolacyjnego zaprojektowanego z użyciem zewnętrznej aplikacji, np. MATLABa,
- deklaracje definiujące bloczki implementowanego algorytmu,
- definicje połączeń z użyciem nazw wejść i wyjść poszczególnych bloczków (skompilowana aplikacja na etapie definiowania połączeń weryfikuje ich poprawność, m.in. zgodność zegarów łączonych bloczków),
- wywołanie procedury sprawdzającej czy wszystkie wejścia bloczków są podłączone oraz procedury generującej schemat blokowy do pliku o rozszerzeniu dot (plik ten można przetworzyć z użyciem narzędzia dot z pakietu Graphviz [14] uzyskując z ten sposób rysunki schematów; przykładowe schematy wygenerowane w ten sposób pokazano na rys. 1),
- uruchomienie algorytmu na zadaną liczbę cykli podanego zegara (nie musi to być zegar główny); kolejne wywołanie polecenia Execute kontynuuje pracę algorytmu; w powyższym przykładzie metoda Execute jest wywoływana dopóki aktywne jest połączenie IP, przez które przekazywany jest przetwarzany sygnał,
- zwolnienie zasobów powiązanych z zegarami algorytmów, w tym automatycznie tworzonych zegarów pochodnych; w przypadku omawianego kodu jest to zegar wyjściowy automatycznie wygenerowany w bloczku DSPu_SamplingRateConversion.

Jak widać definiowanie i łączenie elementów implementowanego algorytmu jest bardzo uproszczone i praktycznie w stosunku jeden do jeden może być powiązane ze schematem blokowym algorytmu. Dodatkowo implementację znacząco upraszczają wbudowane funkcje diagnostyczne wykonywane automatycznie na etapie definiowania połączeń i w trakcie pracy algorytmu oraz wykonywane na życzenie użytkownika. Możliwe jest również łatwe wytworzenie schematu zaimplementowanego algorytmu.

Bibliotekę cechuje również duża otwartość, która umożliwia łatwe rozszerzenie jej możliwości przez użytkownika. W ramach oferowanych przez bibliotekę funkcjonalności możliwe jest wykorzystanie bloczków DSPu_MyFunction oraz DSPu_OutpuBuffer i DSPu_InputBuffer, które pozwalają na dołączenie funkcji obsługi (callback) definiowanej przez użytkownika i wywoływanej przez bibliotekę, gdy konieczne jest przetworzenie próbek wejściowych i wygenerowanie próbek wyjściowych bloczka. W zależności od konfiguracji, bloczki te umożliwiają implementację dowolnych bloczków przetwarzania oraz źródeł sygnału, w tym źródeł sterowanych. Dla bardziej zaawansowanych użytkowników dostępna jest również możliwość uzupełniania biblioteki o własne bloczki pochodne od klas DSP_block lub DSP_source. Kolejną możliwością jest tworzenie klas makropoleceń pochodnych od klasy DSP_macro, w których definiuje się bloczki i połączenia, tak jak przy normalnym definiowaniu algorytmów, z tą różnicą, że dodatkowo określa się wejścia i wyjścia wybranych bloczków wewnętrznych tej klasy, które są udostępniane na zewnątrz. Taką klasę makro, można później wykorzystywać w kodzie do tworzeniu algorytmów, tak jak każdy inny bloczek dostępny w bibliotece, przy czym wewnętrznie biblioteka realizuje każdą instancję takiego makra jako grupę bloczków i połączeń pomiędzy nimi.

4. PODSUMOWANIE

Przedstawiona biblioteka DSPElib umożliwia oprogramowywanie w C++ algorytmów DSP na poziomie struktury blokowej, co jest szczególnie istotne na etapie prototypowania algorytmów. Zwłaszcza przydatna jest łatwość modyfikowania struktury oraz możliwość uzyskania wglądu w reprezentacje sygnałów w dowolnym miejscu schematu poprzez dodanie jednego obiektu reprezentującego wyjście do pliku oraz jego podłączenie do interesującego nas wyjścia. Poza uproszczeniem implementacji algorytmów biblioteka zawiera w sobie rozbudowane mechanizmy detekcji i raportowania błędów zawartych w strukturze algorytmu przetwarzania sygnału. Mechanizmy te są aktywne podczas uruchamianiu aplikacji w trybie debugowania. Przykładowo, procedury biblioteczne wykrywają i raportują, podając nazwy powiązanych z problemem bloków i wyjść, próby podłączenia wielu wyjść do jednego wejścia, pętle sprzężenia zwrotnego nie zawierające elementu opóźniającego, czy też niezgodność zegarów łączonych bloków. Właściwość ta ułatwia zrealizowanie struktury spełniającej zasady realizacji algorytmów DSP. Jednak, ponieważ struktura spełniająca te zasady może nie być zgodna z założeniami programisty, przydatna jest również dostępna w bibliotece funkcja generacji schematów na podstawie utworzonej struktury bloków przetwarzania sygnałów. Zauważmy, że struktura ta niekoniecznie wynika bezpośrednio z samego kodu C++, może być ona bowiem konstruowanych programowo, np. zależnie od parametrów wprowadzanych przez użytkownika aplikacji określającego np. jaki wariant algorytmu przetwarzania należy zastosować.

Poszukując bibliotek C++ dedykowanych implementacji algorytmów cyfrowego przetwarzania sygnałów można głównie natrafić na biblioteki implementujące zestawy zoptymalizowanych narzędzi DSP

do analizy i blokowego przetwarzania sygnałów, np. Jamoma DSP [15], ICST DSP [16] i Aquila [17]. Niektóre rozwiązania, np. SPUC [18], są przystosowane do potokowej realizacji przetwarzania, ale tak jak i w przypadku poprzednio wymienionych bibliotek interakcję pomiędzy bloczkami musi zaimplementować i przetestować użytkownik. Zbliżoną do prezentowanej tutaj biblioteki pod względem użytkowym jest biblioteka slib [19]. Jednak jej możliwości w realizacji algorytmów wieloszybkościowych oraz algorytmów zawierających pętle sprzężenia zwrotnego, są bardzo ograniczone z uwagi na stosowanie buforów służących do gromadzenie próbek przekazywanych pomiędzy bloczkami. W bibliotece tej brakuje też zaawansowanego zarządzania zegarami algorytmów przydatnego w implementacji algorytmów wieloszybkościowych. Kolejną cechą wyróżniającą bibliotekę DSPElib jest rozbudowane raportowanie błędów.

Przedstawiona biblioteka jest wykorzystywana przez jej autora w pracy badawczej oraz dydaktyce. Posługując się biblioteką studenci już po krótkim przygotowaniu mogą samodzielnie implementować, modyfikować i testować autonomiczne aplikacje DSP korzystając z narzędzi OpenSource'owych. Dodatkowo, otwarty charakter biblioteki umożliwia zaawansowanym użytkownikom na uzupełnianie zestawu dostępnych bloczków DSP korzystając z uproszczonych wzorców.

5. BIBLIOGRAFIA

1. Zieliński T.P., Korohoda P., Rumian R.: Cyfrowe przetwarzanie sygnałów w telekomunikacji, Wydawnictwo Naukowe PWN, 2014, ISBN: 978-83-01-17445-3.
2. Blok M.: Algorytm pozyskiwania symboli z synchronizacją symbolową operującą na przebiegu błędu detektora Gardnera, Zeszyty Naukowe Wydziału ETI Politechniki Gdańskiej. Technologie Informacyjne 19, 2010, s. 453-458.
3. Harris F.J.: Multirate signal processing for communication systems, Prentice Hall PTR, 2004.
4. Smith S.W. Cyfrowe przetwarzanie sygnałów DSP. Praktyczny poradnik dla inżynierów i naukowców, Wydawnictwo BTC, 2007.
5. Ingle V. K., Proakis J. G.: Digital signal processing using MATLAB, Cengage Learning, 2016.
6. Eaton J. W.: Octave: Past, present and future, Proceedings of the 2nd International Workshop on Distributed Statistical Computing. 2001.
7. Blok M.: Cyfrowy odbiornik FSK, VII Konferencja Naukowo-Techniczna Systemy Rozpoznania i Walki Elektronicznej KNTWE'08, Warszawa 9-11 grudnia 2008, s. 1-11.
8. Ćwikowski Ł., Blok M.: Educational model of the OFDM modulator and demodulator, Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2009, 75021R.
9. Saramaki T., Neuvo T., Mitra S. K.: Design of computationally efficient interpolated FIR filters, IEEE Transactions on Circuits and Systems 35(1), 1988, pp. 70-88.
10. Vaidyanathan P. P.: Multirate digital filters, filter banks, polyphase networks, and applications: a tutorial, Proceedings of the IEEE 78(1), 1990, pp. 56-93.
11. Blok M.: Edukacyjne narzędzie do badania zjawisk zachodzących podczas konwersji AC/CA, Zeszyty Naukowe Wydziału ETI Politechniki Gdańskiej. Technologie Informacyjne 2010, s. 447-452.
12. Blok M.: Badanie właściwości sygnału telegraficznego oraz sygnału mowy przesyłanych przez kanał analogowy, Zeszyty Naukowe Wydziału Elektrotechniki i Automatyki Politechniki Gdańskiej, 28, 2010, s. 39-42.
13. Gumiński W.: Komputerowy system cyfrowego przetwarzania sygnałów telekomunikacyjnych -DSPS, Krajowe Sympozjum Telekomunikacji'93, Bydgoszcz, t. D, 1993, s. 99-106.
14. Ellson J. i inni: Graphviz - open source graph drawing tools, International Symposium on Graph Drawing, Springer Berlin Heidelberg, 2001, pp. 483-484.
15. Place T., Trond L., Nils P.: A flexible and dynamic C++ framework and library for digital audio signal processing. Proc. of the International Computer Music Conference (ICMC), 2010.
16. Papetti S.: The icst dsp library: A versatile and efficient toolset for audio processing and analysis applications. Proceedings of the 9th Sound and Music Computing Conference, 2012.
17. Paul J.S.: Signal Processing in C++ Using Aquila 3.0, Electronics for you, http://electronicsforu.com/ buyers-guides/software-buyers-guide/signal-processing-c-using-aquila-3-0, 2016.
18. Kirke T.: SPUC - Signal Processing Using C++. wersja 2.4.1, 10.10.2015, https://github.com/ audiofilter/spuc/, dostęp 29.09.2016.
19. Pekar D., Obradović R.: C++ Library for Digital Signal Processing - slib. Proc. IX Telekomunikacioni Forum Telfor, 2001.

DSPElib - C ++ LIBRARY FOR RAPID DEVELOPMENT OF MULTIRATE DIGITAL SIGNAL PROCESSING ALGORITHMS

The C ++ library called DSPElib - Digital Signal Processing Engine library developed by the author is presented in the paper. The library allows for simple implementation of multirate signal processing algorithms containing feedback loops and thus can be used for rapid prototyping of this type of algorithms. Using the library the algorithms can be implemented as stand-alone applications both for Windows or Linux platforms.

Keywords: digital signal processing, multirate signal processing, C++ library, rapid algorithms prototyping.