Osciloscop Software

Abstract

Osciloscopul este un dispozitiv utilizat la analiza semnalelor în diverse ramuri ale industriei precum ingineria electrică, industria constructoare de mașini, medicină, telecomunicații, etc. Deoarece este un dispozitiv care oferă posibilitatea vizualizării formei de undă a tensiunii de pe un circuit electric, el joaca un rol important în înțelegerea și analiza componentelor electrice, precum și a corectarii sau mentenanței acestora. Pentru aplicații care implică utilizarea unor curenți de frecvență redusă sau pentru experimente cu scop didactic, folosirea unui osciloscop fizic (analogic sau digital) poate prea dificilă din punct de vedere al costurilor ridicate sau al transportului acestor dispositive. Lucrarea de față propune rezolvarea acestor probleme prin realizarea unei alternative software de implementare a osciloscopului clasic.

Osciloscopul software implică un cost minim de transport necesar copierii datelor dintre calculatoare, precum și costuri de folosire mult mai scazute datorită ansamblului hardware care se reduce la circuite de protectie la supratensiune, divizoare de tensiune și circuite de interfațare cu placa de sunet de pe dispozitivul ce gazduiește aplicația software. Datorită faptului că majoritatea utilizatorilor dețin deja calculatoare pe care pot rula versiunea de osciloscop software, costul acestor calculatoare nu este luat în considerare. De asemenea, odată ce a fost realizată o implementare a osciloscopului software, aceasta poate fi dezvoltată ulterior pentru a adăuga diferite funcționalități ce servesc unor scopuri particulare tipice unui grup restrans de utilizatori, în funcție de necesitățile acestora, dar pe care un osciloscop clasic analogic nu le indeplinește. De exemplu: generarea de semnal cu anumiți parametrii ajustabili, analiza semnalului achiziționat în frecvență, aplicarea de diverse filtre, înregistrarea și partajarea datelor, etc.

Unul dintre cele mai importante avantaje ale realizării unui osciloscop software il reprezintă posibilitatea de a livra utilizatorilor versiuni îmbunătățite ale acestuia ceea ce implică dezvoltarea lui ulterioară, aceasta putând fi realizată de autorul inițial sau de contribuitori la proiectul existent. Aceștia pot asigura o continua îmbunătățire a aplicației ce are în vedere pe langă adăugarea de funcționalități noi, pastrarea acelora deja existente, adăugarea optimizărilor de performanță atunci cand acestea sunt necesare, menținerea compatibilității cu noile platforme și sistemele de operare, dar nu în ultimul rând, menținerea unui produs cu o fiabilitate mai ridicată decât osciloscopul fizic.

În vederea respectării avantajelor menționate mai sus, lucrarea de față oferă pe lângă o versiune de implementare a osciloscopului software, principiile de construcție ale acestuia, dificultățile care pot apărea în dezvoltarea unei astfel de aplicații și rezolvarea acestora, limitările pe care le are implementarea software bazată pe achiziția de date prin intermediul portului audio și modul în care se poate realiza interfațarea software cu placa audio pentru obținerea celor mai avansate performanțe pe care aceasta le oferă. După parcurgerea acestei lucrări, un dezvoltator va dispune de o bază de informații solidă ce îl va ajuta atât în lucrul cu această aplicație, cât și în dezvoltarea ei ulterioară sau chiar și în dezvoltarea unei versiuni diferite de osciloscop software, având aceleași funcționalități și structură modulară ce va suporta noi dezvoltări. Realizarea unei aplicații modulare este importantă în vederea dezvoltărilor ce pot fi realizate de mai multi oameni simultan, și a menținerii aplicației scalabile și flexibile, de exemplu funcționalități precum partajarea datelor și aplicarea filtrelor de frecvență pot fi implementate în paralel fără a se influența reciproc.

Suportul Hardware

Majoritatea calculatoarelor dispun de placi de sunet cu rate de eșantionare de pana la 192 kHz, și interfețe audio stereo de tip jack, ceea ce împreună cu puterea de procesare a majoritatea dispozitivelor, furnizează suportul hardware ideal realizării osciloscopului software cu două canale care suportă frecvențe ale semnalului de intrare de până la 20 kHz. Conform teoremei lui Shanon se poate face analiza semnalelor de intrare cu frecvențe de maxim 97 kHz, fară a se pierde informații la reconstrucția semnalului eșantionat, prin interpolarea eșantioanelor, însă pentru aceste semnale, reconstrucția nu va fi de o fidelitate foarte ridicată din cauza pierderii anumitor detalii. Se poate încerca pentru o reproducere mai fidelă a semnalului interpolarea eșantioanelor folosind o funcție polinomială de gradul doi, dar această metodă nu garantează producerea unor grafice de o acuratețe mai ridicată în fiecare caz.

Considerând structura simplă a unui port jack de tip stereo fară microfon, se poate conecta cu ușurință un circuit de achiziție de date.



 Pe langă frecvența semnalului de intrare, placile de sunet pentru uz general au ca parametri suportați o tensiune de intrare ce poate varia de la +0.25V la -0.25V, cu o adâncime de bit a convertorului analog digital de 16 sau 24 de biți, ceea ce oferă posibilitatea unei reprezentări precise a semnalului achiziționat, chiar și în cazul utilizării adâncimii de bit de 16 biți.

Suportul software

În funcție de modul în care diverse aplicații software au fost utilizate în dezvoltarea proiectului curent, acestea se pot grupa în:

- Limbaj și platforma de dezvoltare

- Medii de dezvoltare și compilare;

- Librarii referentiate de aplicație;

- Utilitare de versionare.

Limbaj și platforma de dezvoltare

Aplicația a fost dezvoltata în limbajul C++. Acesta este un limbaj orientat pe obiect ce permite realizarea de structuri complexe, ierarhii de clase ce au comportament polimorfic, asemeni unui limbaj de nivel înalt, dar de asemenea permitand lucrul cu memoria asemeni unui limbaj de nivel jos. Acestea îl fac ideal pentru dezvoltarea aplicatiilor mari, scalabile și modulare, dar totodata el ofera performanțe ridicate datorită eficientei în lucrul cu memoria și tipurilor de date primare ce nu presupun un nivel ridicat de abstactizare astfel încât în cele mai multe cazuri este folosit un numar minim de cicluri de ceas.

Medii de dezvoltare și compilare

În dezvoltarea aplicației s-a folosit mediul de dezvoltare integrat Visual Studio de la Microsoft, versiunea 2012 Express, și versiunile Community 2015 și 2017, întrucât dezvoltarea acesteia a avut loc iterativ iar versiunea din urma de Visual Studio ofera un mai bun suport pentru dezvoltarea în C++ precum și performanțe mărite în compilarea aplicatiilor scrise în acest limbaj de programare. Având în vedere faptul ca versiunile Express și Community sunt versiuni gratuite pentru dezvoltarile care nu sunt de tip comercial, utilizarea lor devine ideala pentru acest tip de proiect. Visual Studio ofera un mediu flexibil de programare pentru a gestiona atât aplicații mici cât și aplicații cu o complexiatate ridicată ce inglobeaza un numar mare de proiecte de diferite tipuri, cu diverse setari de compilare și dependente paricularizate pentru fiecare în parte.

În cadrul acestei implementari, osciloscopul software a fost dezvoltat și compilat sub sistemul de operare Windows, folosind compilatorul dedicat dezvoltat de compania Microsoft, „Microsoft C++ Compiler (MSVC)”. Uilizarea limbajului C++ fără integrarea în aplicație a librariilor care depind de acest sistem de operare sunt factori care mențin aplicația compilabila pe alte platforme precum Linux, prin recompilarea cu compilatoare GNU precum G++ sau GCC.

Librarii referentiate de aplicație

În afara de containerele și librariile din librarila standard de C++, cunoscută și sub numele de „C++ Standard Library” sau „stdlib” care sunt deja integrate în mediul de dezvoltare oferit de Visual Studio, au fost folosite librariile „wxWidgets” pentru crearea interfetei grafice și „PortAudio” pentru accesarea functiilor oferite de placa de sunet a mașinii pe care aplicația ruleaza. Libraria standard de C++ a fost folosită pentru a dispune de containere generice precum vectorii din namespace-ul „std” sau pentru lucrul cu fire de execuție separate. Atât cele doua mari librarii, „wxWidgets” și „PortAudio”, precum și libraria standard de C++ sunt independente de platforma pe care lucrează, astfel încât s-a avut în vedere mentinerea portabilitatii aplicației, dar și performanța ridicată prin suportul pentru limbajul de programare de nivel jos.

Utilitare de versionare

Mentinerea codului sursa al aplicației într-un singur loc prezinta riscuri din cauza centralizarii datelor și a potențialului defectarii din diverse cauze a mașinii pe care acestea sunt stocate, ceea ce rezulta în pierderea aplicației. Utilizarea unor apicatii software de versionare se impune în dezvoltarea proiectelor de dimensiuni ridicate, dar și în cazul acelora a caror date prezinta o importanța deosebita. Astfel s-au folosit pentru versionarea fisierelor sursa pe mașina locala, versiunea de „Mercurial” a utilitarului „Tortoise”, cunoscută și sub numele de „TortoiseHg” și crearea unui container de tip „Mercurial”. Acesta a fost urcat pe un repository remote pe platforma „Bitbucket”, aceasta prezentand avantajul securitatii datelor, dar și posibilitatea accesarii acestora din diferite locatii, independent de mașina pe care se dorește compilarea și rularea aplicației.

Versionarea aplicației cu astfel de utilitare presupune numeroase avantaje precum eficiența ridicată în eliminarea defectelor aparute în dezvoltare, prin compararea cu ușurința a diferitelor versiuni de aplicație și localizarea secventelor de cod problematic. Printre avantajele utilizarii unor aplicații de versionare se număra și posibilitatea mentinerii în paralel a diferite variante a proiectului ce trebuie dezvoltat sau lucrul în paralel a mai multor dezvoltatori, deși acesta nu a fost aplicabil în cazul aplicației curente.

Structura aplicației

Aplicația a fost împarțită în două module, pe baza responsabilității acestora, unul care se ocupă cu achiziția de date iar altul cu prezentarea acestora și interacțiunea cu utilizatorul. Modulul de achiziție de date a fost compilat sub forma unei librarii dinamice (Dynamic Link Library) în timp ce modulul de afișare de date a fost compilat sub forma unui executabil, astfel încât aplicația poate fi rulată lansând direct fișierul executabil. Comunicarea dintre cele două module se face prin expunerea de funcții din librăria pentru achiziție de date, acestea fiind apelate de către modulul de interfață grafică în funcție de setările făcute de utilizator.

Comunicarea dintre module

Pentru a se putea realiza comunicația dintre cele două module, acestea trebuie să conțină un antet (header) sau un set de antete comune, astfel încăt la compilare să se cunoască semnăturile funcțiilor expuse de librăria dinamică, ce pot fi apelate de modulul de afișare de date. Acestea se pot schimba în decursul dezvoltării, sau cu noile versiuni de aplicație, deși în cazul în care se realizează schimbarea semnăturii funcțiilor, sau înlocuirea completă a acestora este contraindicat în mediile comerciale, deoarece acest lucru duce la incompatibilitatea fișierelor binare ce pot fi utilizate de mai mulți clienți. În cazul în care aplicația este dezvoltată de un număr redus de contribuitori, sincronizarea dintre acestia este relativ facilă, însă în cazul în care pe parcursul dezvoltării proiectului, apare un număr mare de clienți ai librăriilor, schimbarea funcțiilor expuse de acestea poate avea costuri foarte ridicate, preferându-se menținerea funcțiilor vechi și adăugarea de noi funții cu semnături schimbate (parameter overloading). În aceste cazuri, deciziile privind arhitectura proiectului devin dificile, ele însumând un număr ridicat de factori, printre care cei economici, deoarece menținerea funcțiilor duplicate poate avea un impact semnificativ în timpul de dezvoltare al aplicației, cât și factori sociali, deoarece clienții librăriilor schimbate își pierd încrederea în echipele care se ocupă de dezvoltarea acestor proiecte.

În implementarea proiectului curent atât semnăturile funcțiilor cât și funcțiile în sine au fost schimbate, pentru diverse adăugări sau schimbări de funcționalități, ajungând ca la momentul redactării documentului de față, acestea să aibă următoarele declarații:



Funcțiile de “Initialize” și “GetSampleRate” sunt apelate la pornirea aplicației, pentru inițializarea canalului de achiziție a datelor, iar rata de eșantionare este necesară pentru calculul frecvențelor rezultate în urma aplicării transformatei Fourier în cadrul modulului de afișare de date. Celelalte funcții sunt folosite strict pentru colectarea datelor. Se poate observa duplicarea acestor funcții, cu având diferit în denumire doar cuvintele “Left” și “Right” întrucât acestea se referă la datele de pe canalul de eșantionare stâng respectiv drept. S-a realizat și separarea în funcție de destinația datelor, acestea fiind pentru semnalul afișat pe ecran sub forma de semnal achiziționat pentru analiza făcută pe osciloscop, dar și ca semnal ce este folosit pentru analiza în frecvență. Deși aparent este vorba de aceleasi esantioane pentru fiecare canal și nu ar trebui făcută această separare, datele achiziționate pentru construirea spectrului de frecvență nu au nevoie de verificarea unui prag de eșantionare (threshold) precum în cazul celor afișate în primă instanță pe osciloscop, iar înlăturarea acestui prag de eșantionare îmbunătățește performanțele aplicației.

Deoarece datele prelevate pentru construirea spectrului de frecvență nu necesită verificarea pragului de eșantionare, funcțiile expuse pentru obținerea acestor date nu conțin decât un singur parametru, acela fiind numarul de eșantioane dorite de către clientul librăriei. Cazul funcțiilor ce expun datele necesare constuirii formei de undă ce corespunde semnalului primit pe placa de sunet este diferit, întrucât acestea au nevoie și de pragul eșantionare ales de utilizator, astfel încât acestea acceptă parametrii de tipul pointer către obiecte din clase specifice, ce pot conține mai multe preferințe de eșantionare. De asemenea, tot pe principiul datelor compuse, fiecare dintre funcțiile de achiziție de date, returnează adrese ale obiectelor care conțin datele corespunzătoare apelului realizat de client.

Structura claselor care conțin datele schimbate între module este relativ simplă, aici însă este posibilă folosirea comportamentului polimorfic al limbajului de programare, pentru obiectele fiind declarate de tip clasa cu metode pur virtuale sau cum este popular ăn limbajele complet orientate pe obiect, clasa abstracta.

Avem astfel clasa “IdataRequest” a cărei implementare se află în modulul de afișare de date:



Și clasa “IDataResponse” a cărei implementare este în modulul de achiziție de date:



Fiecare modul este independent de implementarea pe care celălalt o oferă clasei abstracte ce îi corespunde, atât timp cât cele două module păstrează semnături comune ale acestor clase. Denumirile claselor precedate de vocala “I” provin de la cuvântul “Interface” și marchează un standard pentru a marca tipuri ce reprezină abstractizări ale claselor sau, cu alte cuvinte, contracte pe care acestea urmează să le îndeplinească, dar întrucât interfețele nu sunt disponibile în limbajul C++, a fost păstrat numai standardul de denumire specific. Implementarea acestei aplicații într-un limbaj de programare complet orientat pe obiect ar trebui să permită sau chiar să încurajeze utilizarea interfețelor în locul claselor abstracte în cazul de față.

Modulul de achiziție de date

Inițializare

Librăria PortAudio necesită inițializarea unui flux de achiziție de eșantioane cu unul din dispozitivele de intrare audio, iar pentru această aplicație, în cazul în care există mai multe dispozitive de acest fel, este ales cel implicit. Pentru a nu exista momente în care aplicația nu este funcțională de a lungul rulării din cauza deconectării dispozitivului audio pe care se ascultă, acest modul deține un fir de execuție dedicat verificării fluxului de date și reinițializării acestuia în cazul unei deconectări. Tot la inițializare se verifică toate frecvențele de eșantionare disponibile și se alege cea mai mare dintre acestea pentru a beneficia de utilizări ale aplicației cu frecvențe de intrare cât mai ridicate. Ratele de eșantionare care sunt verificate variază de la 8000 de hertzi până la 192kHz și în cazul majorității dispozitivelor este aleasă ultima opțiune.

Tot la inițializare se setează diverși parametri pentru fluxul cu ajutorul căruia se face achiziția de eșantioane, precum numărul de valori care vor fi incluse în cadrul unui buffer cu eșantioane, se alege numărul de canale pe care se face achiziția de date, în cazul aplicației curente acesta fiind doi și se setează funcția de callback. În cazul în care inițializarea a reușit fără a apărea o eroare pe parcursul acestei funcții, se pornește firul de execuție care verifică disponibilitatea canalului de pe care se preiau date și se returnează valoarea adevarat. În cazul deconectării acestui canal, trebuie reapelată întreaga funcție de inițializare, deoarece ascultarea de la un alt dispozitiv de intrare poate presupune setarea unor parametri complet diferiți, dedicați acestuia.

Funcția de callback

Este o funcție ce are o semnătură specifică, fixă, a cărei adresă este trimisă la fluxul de pe care se preiau date cu librăria PortAudio și care este apelată de această librărie atunci când sunt noi date disponibile, de acolo și numele de “callback”. Apelul funcției de callback se poate face cu o frecvență mai ridicată sau mai scăzută, în funcție de dimensiunea aleasă a buffer-ului în care sunt scrise eșantioanele de la placa de sunet. Această valoare este aleasă încă de la inițializarea fluxului de date, dar ea este de asemenea primită ca parametru în semnătura funcției. Pentru a beneficia de o latență cât mai mică în rularea aplicației, dar și de pierderea a cât mai puține eșantioane, pentru acest parametru s-a ales o valoare mică, de opt eșantioane, dar aceasta putând diferi de la o implementare la alta, în funcție de necesitățile fiecărei aplicații și de preferințele dezvoltatorului, în cazul în care după mai multe teste, se descoperă faptul că o valoare mai mare sau mai mică aduce un câștig semnificativ de performanță.

Cu toate că semnătura acestei funcții nu poate fi modificată, implementarea sa rămâne la latitudinea dezvoltatorului, astfel încât datele provenite de la librărie pot fi folosite în orice fel. Există de asemenea sugestii privind modul de implementare al acestei funcții și anume evitarea de rularea a operațiilor costisitoare din punct de vedere al timpului de execuție în cadrul funcției de callback. Deoarece această funcție va fi apelată de fiecare dată când buffer-ul de eșantioane va fi completat, pentru o funcționare ideală, lipsită de erori, se recomandă evitarea operațiunilor precum alocarea dinamică sau dealocarea unei zone mari de memorie, scrierea sau citirea de pe disc, scrierea de informații în consolă au utilizarea mecanismelor ce pot bloca firul de execuție precum mutex-uri, semafoare sau apelarea metodelor ce pun în așteptare firul curent de execuție.

Având în vedere setările utilizate în aplicația curentă, ce pot influența frecvența de apelare a funcției de callback, precum dimensiunea redusă a buffer-ului cu valori de pe placa de sunet, dar și rata de eșantionare ridicată, operațiile care sunt posibile în aceasă funcție sunt restrânse, de aceea se realizează numai adăugarea de eșantioane în buffere, după aplicarea unui factor de conversie de pentru obținerea valorilor cuantizate pe 16 biți, ceea ce presupune împărțirea printr-o valoare de doi la puterea șaisprezece. Deoarece s-a ales inițializarea unui flux pe două canale de achiziție de date, eșantioanele prezente în buffere-ul primit ca parametru în funcția de callback va conține eșantioane intercalate pentru cele două cananle. Pentru stocarea acestora, buffer-ul este iterat cu un indice a cărui valoare este incrementată din doi în doi, prima valoare fiind stocată în containerul pentru canalul stâng iar a doua valoare în containerul pentru canalul drept. Reprezentarea containerelor în memorie este abstractizată în contextul funcției de callback, deoarece aceasta nu este importantă aici, atât timp cât containerul oferă performanțe ridicate la scrierea datelor, ci este utilizată aici tot o clasă abstractă cu metode pur virtuale pentru adăugarea valorilor. Un pointer către o instanță a acestei clase este obținut în interiorul funcției de callback prin apelul unei metode statice, ce reprezintă o implementare a șablonului “Singleton”. Apelul funcției statice a fost utilizat întrucât nu parametrii funcției de callback nu pot fi modificați, dar existența unui container static prealocat este favorabilă pentru performanța aplicației.

Managementul bufferelor

Pentru abținerea unei structuri simple din punct de vedere a algoritmilor utilizați, s-a ales abstractizarea buffer-elor pe mai multe nivele de la cel mai jos, care implică lucrul direct cu zonele de memorie, până la cel mai înalt nivel care expune doar metode pentru inserarea și accesarea datelor, ascunzând alternarea buffer-elor, accesul la pozițiile de citire diferit de accesul la pozițiile de scriere și circularitatea buffer-elor ce previne posibilitatea depășirii spațiului disponibil în buffer. În cadrul acestei secțiuni se va prezenta structura fiecărui nivel de abstractizare a buffer-elor folosite, pornind cu cel mai jos nivel de abstractizare, unde memoria va fi accesată direct, până la cel mai înalt nivel, pentru a se putea urmări complexitatea pe care această structură o ascunde.

Nivelul 1. Buffer-ul circular

Buffer-ul circular este cea mai de jos clasă folosită în cadrul aplicaței curente pentru management-ul memoriei si constituie un container ce poate suporta doar adăugarea de noi valori, citirea numărului de valori noi și citirea fiecărei valori iterativ. Utilizarea unei astfel de structuri simplifică lucrul cu buffer-ul ales efectiv și permite înlocuirea acestuia într-un mod facil, trecerea de la o zonă de memorie alocată dinamic reprezentând un vector de tipuri primitive pentru stocarea valorilor, la liste, vectori sau alte containere specifice din librăria standard de C++ putând fi făcută foarte ușor.

În implementarea curentă, buffer-ul circular deține un vector de valori de tip float, alocat dinamic, al cărei dimensiuni poate fi setată din constructor ca parametru implicit. Folosirea valorii setate în cadrul constructorului a fost preferată pentru facilitarea scrierii de teste unitare, pentru a asigura o funcționare corectă și fiabilitate containerului. Au fost alese date de tip float deoarece calculele efectuate cu acest tip de date au loc mai rapid decât calculele cu valori de tip double, dar și din cauza faptului că spațiul ocupat de acestea este de două ori mai mic. În cazul unui singur container de dimensiuni reduse diferența de spațiu de memorie ocupat de aplicație nu este considerabilă, însă în cazul untilizării mai multor containere de dimensiuni extinse, optimizările de acest fel contribuie semnificativ la rularea aplicației.

Printre datele private ale clasei se numără și dimensiunea buffer-ului. Chiar dacă inițial dimensiunea buffer-ului nu este necesară decât pentru alocarea acestuia, la dealocare dimensiunea buffer-ului nefiind necesară, ulterior aceasta este folosită pentru a nu se adăuga sau citi date de la poziții mai mari decât dimensiunea alocată. Dimensiunea containerului este de asemenea utilizată pentru a determina numărul de valori pe care acesta le deține, în cazul în care containerul este complet ocupat cu valori.

Pentru determinarea numărului de valori disponibile în container se reține poziția pe care s-a adăugat ultima valoare. Această valoare este incrementată cu fiecare adăugare de noi valori iar în cazul în care această valoare depășește dimensiunea containerului, ea este reinițializată cu valoarea zero. Tot la depășirea dimensiunii containerului, o variabilă de tip boolean este setată, astfel încât să se marcheze faptul că dimensiunea maximă a buffer-ului a fost atinsă și următoarele valori vor suprascrie valorile adăugate inițial în buffer. În acest mod, buffer-ul circular nu va face overflow și nu se vor adăuga niciodată valori pe poziții din afara zonei de memorie alocată.

Pentru parcurgerea valorilor din buffer-ul circular, este pus la dispoziție operatorul de iterare “[]” care primește ca parametru indexul pe care se face accesarea de date. În implementarea acestui operator, se testează valoarea booleană pentru a se verifica dacă s-a depășit numărul maxim de valori adăugate în buffer sau dacă încă mai sunt locuri disponibile. Această verificare se face deoarece se dorește accesarea datelor în ordine cronologică adăugării lor în container, astfel încât datele cele mai vechi sunt suprascrise, iar datele citite din container trebuiesc accesate plecând de la prima poziție la care există valori disponibile. În cazul în care nu s-a depășit dimensiunea containerului, accesarea datelor este similară cu accesarea datelor într-un container liniar, astfel prima poziție de pe care se citesc valori este poziția zero, iar accesul acestora se face după pozițiile dorite de utilizatorii containerului. În cazul în care buffer-ul circular are adăugate mai multe valori decât dimensiunea acestuia, atunci citirea de date se face pornind de la ultima poziție pe care au fost adăugate valori plus unu. Așadar, pentru determinarea indexului de acces al valorilor din vector la un anumit index dorit, în cazul în care au fost adăugate date peste limia buffer-ului circular, se folosește următoarea formulă:

k = (i + p) % d

Unde k este poziția pe care se accesează datele, i este indexul pe care utilzatorul exterior clasei îl vede ca fiind poziția de iterare prin buffer, p este ultima poziție pe care s-au adăugat date noi, incrementată cu unu, iar d este dimensiunea buffer-ului circular. Procentul reprezintă operația “modulo” sau restul împărțirii la un anumit număr. Astfel se asigură că niciodată nu se va depăși dimensiunea buffer-ului ca poziție de pe care se face citirea din acesta. Singurul tip de valori cu care se poate accesa buffer-ul pentru care este cauzată o exceptie îl constituie valorile negative, astfel încât este foarte puțin probabil ca utilizarea acestui container customizat să cauzeze erori în rularea aplicației.

Printre operațiile puse la dispoziția clienților acestei clase se numără determinarea cantității de valori disponibile în buffer sau operația de resetare de buffer. Implementarea acestor funcții se face simplu, folosind numai indexul pe care se adaugă date și valoarea booleană care corespunde depășirii dimensiunii maxime a buffer-ului. Astfel resetarea buffer-ului presupune doar asignarea valorii de fals logic variabilei de tip boolean pentru depășirea mărimii buffer-ului, și reinițializarea cu valoarea zero, a variabilei ce reține indexul pe care urmează să se adauge următoarea valoare. Deoarece buffer-ul nu este niciodată golit sau valorile din interiorul acestuia complet reinișializate, se obține o performanță ridicată pe operația de resetare, această optimizare fiind făcută în detrimentul protecției la accesul de poziții invalide. În cazul în care se accesează o valoare de pe o poziție invalidă, mai mare decât numărul de valori introduse în buffer, atunci valoarea returnată va fi una veche, deci responsabilitatea pentru validarea pozițiilor acesate revine clienților clase.

Determinarea numărului de valori disponibile sau introduse în buffer fără resetarea acestuia se face verificând dacă numărul valorilor adăugate a depășit numărul valorilor pe care îl suportă instanța curentă a buffer-ului. În cazul în care au fost introduse mai multe valori decât dimensiunea buffer-ului, atunci numărul de valori disponibile returnat este dimensiunea buffer-ului, altfel se returnează indexul la care s-a adăugat ultima valoare incrementat cu unu.

Nivelul 2. Buffer-ul de răspuns

Buffer-ul de răspuns este o clasă ce implementează clasa abstractă “IDataResponse” și este utilizat ca obiect returnat de către modulul de achiziție de date în cadrul unui răspuns la o solicitare de date. Acest tip de date reprezintă un al doilea nivel de abstractizare a vectorului în care sunt ținute datele, având obiectivul de a abstractiza logica pentru a oferi clienților un container cu un anumit număr de eșantioane care pot fi numărate ținând cont sau nu de un prag de eșantionare, care pot fi alese crescător sau descrescător față de pragul ales. Acest nivel de abstractizare implementează funcții specifice noțiunii de achiziție de date corespunzătoare unui semnal, iar din acest motiv, în secțiunea următoare se va folosi termenul de eșantion sau valoare achiziționată, și nu doar date dintr-un container.

Acest container deține un număr redus de câmpuri, printre care un pointer la un buffer circular descris în secțiunea anterioară, un câmp de tip întreg ce corespunde spațiului dorit pe care trebuie să îl aibă container-ul ca răspuns al solicitării curente de date, poziția de acces în container, care este relativă la primul eșantion care depășește pragul de eșantionare din apelul curent, și un câmp de tip boolean pentru a se păstra semnul pantei pe care semnalul eșantionat ar trebui să o dețină. Dintre aceste date numai valoarea booleană corespunzătoare semnului pantei semnalului eșantionat nu este o dată necesară a containerului, ea putând fi primită în cadrul fiecărui apel de metodă care solicită date. Alegerea acestei valori ca făcând parte din datele membre a clasei curente a fost făcută din motive de performanță, deoarece ea va fi schimbată de rar în comparație cu numărul de cereri de date care sunt făcute la acest modul. În alte implementări ale aplicației curente sau ale acestui container, este recomandat ca să se limiteze numărul de date membre, fiind favorizat transferul de date prin obiecte primite ca parametru la cererea curentă.