UNIVERZA V LJUBLJANI
Fakulteta za elektrotehniko

Marko Đorić

AVTOMATIZACIJA LABORATORIJSKIH INŠTRUMENTOV V PROGRAMSKEM JEZIKU C++

DIPLOMSKO DELO UNIVERZITETNEGA ŠTUDIJA

Mentor: doc. dr. Boštjan Murovec

Ljubljana, april 2012

Zahvala

V prvi vrsti bi se rad zahvalil svojem mentorju doc. dr. Boštjanu Murovcu za dobro usmerjanje, koristne nasvete, podporo in potrpljenje.

Velika zahvala gre družini, ki mi je omogočila študij in me pri tem podpirala od začetka do konca. Zahvaljujem se tudi dekletu za razumevanje, pomoč in vzpobudne besede.

Povzetek

Zajemanje karakteristik električnih vezij z večanjem števila parametrov in inštrumentov postane časovno zamudno. Programski jeziki nam omogočajo izdelavo programov, ki čas izvajanja meritev znatno zmanjšajo. V našem primeru smo tri laboratorijske merilne inštrumente abstrahirali v C++ razrede, ki nam omogočajo avtomatizirano upravljanje teh inštrumentov.

Ustreznost razredov smo preizkusili na primeru demonstracijskih merilnih aplikacij.

Ključne besede: programski jezik C++, programska abstrakcija merilnih inštrumentov, frekvenčni spekter, statistika, porazdelitev, amplitudna karakteristika, popačenje

Abstract

Time needed to acquire characteristics of electrical circuits is increasing by the number of parameters and instruments included in measurement. Programming languages allows us to develop programs that significantly reduce the time needed for measurements. We have developed abstraction in the form of C++ classes for three laboratory instruments, which allow us to remotely control those instruments.

We have tested classes adequacy by developing demonstrational measurement applications.

Keywords: C++, frequency spectrum, statistics, distribution, amplitude characteristic, distortion

Kazalo

Kazalo slik

1 Uvod

V današnjem času povpraševanje po avtomatiziranih sistemih konstantno narašča. Dosedanje ročno delo vse pogosteje opravljajo različni programi, kar omeji vpliv človeškega faktorja in vpelje računalniško natančnost. Sodobnejši merilni inštrumenti imajo možnost povezave z osebnim računalnikom preko različnih vodil (USB, RS-232, GPIB, Ethernet), kar omogoča njihovo upravljanje na daljavo (ang. remote control) in avtomatizacijo delovanja.

Inštrumente uporabljamo v pedagoške namene kot tudi pri znanstvenih in razvojnih projektih. Ker je ročno nastavljanje parametrov in odčitavanje vrednosti zamudno, smo se odločili razviti C++ razrede, ki te inštrumente abstrahirajo in omogočajo hiter razvoj merilnih aplikacij bez poznavanja tehničnih podrobnosti inštrumentov in njihove komunikacije z osebnim računalnikom.

Nastale C++ razrede smo preizkusili z izdelavo naslednjih demonstracijskih programov. Za inštrument Hameg HM8118 (Poglavje 2.1) smo razvili dva demonstracijska programa. Prvi prikazuje frekvenčno odvisnost pasivnih elementov električnega vezja, medtem ko drugi omogoča izračun statističnih vrednosti in določanje porazdelitve rezultatov meritev.

Tretji program omogoča zajemanje amplitudne karakteristike pri različni obliki in frekvenci vhodnega signala z inštrumentom Keithley 2016p (Poglavje 2.3).

Zadnja dva programa prikazujeta kombinirano delovanje dveh inštrumentov: napajanika TTi QL355TP (Poglavje 2.2) in inštrumenta Keithley 2016p. Prvi program nam omogoča zajem spektra napetostnega signala s parametrizacijo amplitude vzbujanja. Drugi program prikazuje odvisnost popačenja vhodnega signala od napajalne napetosti vezja. Z inštrumentom Keithley 2016p generiramo vzbujevalni signal in zajemamo rezultate meritev. Napajalno napetost generiramo z usmernikom TTi QL355TP.

Vsi programi imajo možnost shranjevanja rezultatov meritev v tekstovne datoteke, kar omogoča njihovo nadaljnjo obdelavo. Razvoj vseh omenjenih komponent je potekal v razvojnem okolju Microsoft Visual Studio 2008.

2 Uporabljeni inštrumenti

V tem poglavju predstavljamo inštrumente, s katerimi izvajamo avtomatizacijo meritev. Za vsak inštrument smo našteli njegove glavne lastnosti.

2.1 Hameg HM8118

Hameg HM8118 (Slika 2.1) [1,2] omogoča meritev parametrov pasivnih elementov, kot so upornost, kapacitivnost, induktivnost, impedanca ali admitanca, s frekvenco 12 meritev v sekundi in maksimalno natančnostjo 0,05 %.

Slika 2.1 : Hameg LCR Bridge HM8118 (preslikano iz [1])

LCR meter HM8118 pri vsaki od 9 merilnih funkcij meri in prikazuje dva parametra hkrati. Lasnosti inštrumenta, kot so izbira merilne napetosti (0,1 Vrms – 1,5 Vrms) in možnost vsiljevanja DC napetosti ali toka na merilni AC signal, omogočajo zelo učinkovito ocenitev vseh LCR parametrov. Inštrument ima možnost nastavljanja frekvence v območju od 20 Hz do 200 kHz. Z računalnikom ga upravljamo preko vodil USB, RS-232 in GPIB.

2.2 Napajalnik TTi QL355TP

TTi QL355TP (Slika 2.2) [3,4] je dvokanalni napajalnik z ločljivostjo nastavljanja napetosti do 1 mV tudi pri polni napetosti na izhodu (35 V). Ima možnost nastavljanja treh različnih območij toka pri nižjih napetostih:

- 0 do 35 V in 0 do 3 A,
- 0 do 15 V in 0 do 5 A,
- 0 do 35 V in 0 do 500 mA.

Tipalne (ang. sense) priključne sponke inštrumenta so ključnega pomena za natančno meritev napetosti, ker bistveno zmanjšajo napako zaradi padca napetosti na napajalnih linijah.

Slika 2.2 : TTi QL355TP Power Supply (preslikano iz [3])

Popolnoma nastavljiv napetostni in tokovni prožilnik preobremenitve (ang. over-voltage trip, over-current trip) ter prožilnik temperaturne preobremenitve zagotavljajo celostno zaščito inštrumenta v primeru napačne uporabe.

QL355TP upravljamo z računalnikom preko vodil USB, RS-232 in GPIB.

2.3 Keithley 2016p

Keithley 2016p (Slika 2.3) [5,6,7] uporablja digitalni signalni procesor, ki zagotovi natančno analizo frekvenčnega spektra pri merjenju nelinearnega popačenja. Meri harmonsko popačenje (THD, THD+N, SINAD) [12] v frekvenčnem območju od 20 Hz do 50 kHz.

Slika 2.3 : Keithley 2016p (preslikano iz [5])

Totalna harmonska distorzija ali THD (enačba 1) je definirana kot razmerje med vsoto moči višjih harmonskih komponent signala in močjo nosilnega (vzbujevalnega) sinusnega signala. Pri predpostavki, da se vse moči trošijo na isti impedanci, smemo razmerje moči prevesti na razmerje napetosti. Tako obliko prikazuje enačba (1).

$$THD = \frac{\sqrt{(V_2^2 + V_3^2 + \cdots + V_i^2)}}{V_1} \tag{1}$$

Veličina V_1 predstavlja amplitudo nosilnega signala, veličine V_2, $V_3 \ldots V_i$ pa predstavljajo amplitudo harmonskih komponent signala.

Totalna harmonska distorzija s šumom ali THD+N (2) je definirana podobno kot THD le-da, poleg harmonskih komponent signala, v razmerje vključimo še šum, ki je prisoten v opazovanem frekvenčnem območju izhodnega signala. Ker merjenje popačenja brez šuma zahteva merjenje posameznih harmonskih komponent signala, namesto njegove celotne moči,

ob analognem filtriranju osnovnega harmonika, je THD+N lažja in bolj razširjena meritev. Veličina *N* predstavlja napetost šumnega dela signala.

$$THD + N = \frac{\sqrt{(V_2^2 + V_3^2 + \cdots + V_{1024}^2 + N^2)}}{V_1} \quad (2)$$

Razmerje signal-šum in distorzija ali SINAD (3), je definirano kot razmerje med močjo celotnega signala in močjo višjih harmonskih komponent brez nosilnega signala. Enačba je tudi tu prevedena na napetostno obliko.

$$SINAD = \frac{\sqrt{(V_2^2 + V_3^2 + \cdots + V_{1024}^2 + N^2 + V_1^2)}}{\sqrt{(V_2^2 + V_3^2 + \cdots + V_{1024}^2 + N^2)}} \quad (3)$$

Inštrument ima na zadnji strani sinusni vir z nastavljivo amplitudo in frekvenco ter sekundarni izhod, ki oddaja invertiran (fazno zamaknjen za 180°) sinusni signal.

Keithley 2016p je tudi zelo natančen multimeter, saj omogoča merjenje napetosti, toka ali frekvence na 6½ decimalnih mest ter merjenje upornosti na 4 decimalna mesta.

Z uporabo zunanjega krmiljenja lahko zajamemo posamezne amplitude frekvenčnega spektra merjenega signala, kjer je najnižja možna merjena frekvenca 20 Hz, najvišja pa 50 kHz. V eni meritvi lahko zajamemo amplitudo spektra pri do 1023 različnih frekvencah. Z inštrumentom merimo neposredno amplitudo pri določeni frekvenci ali pa spremembo amplitude med različnimi frekvencami.

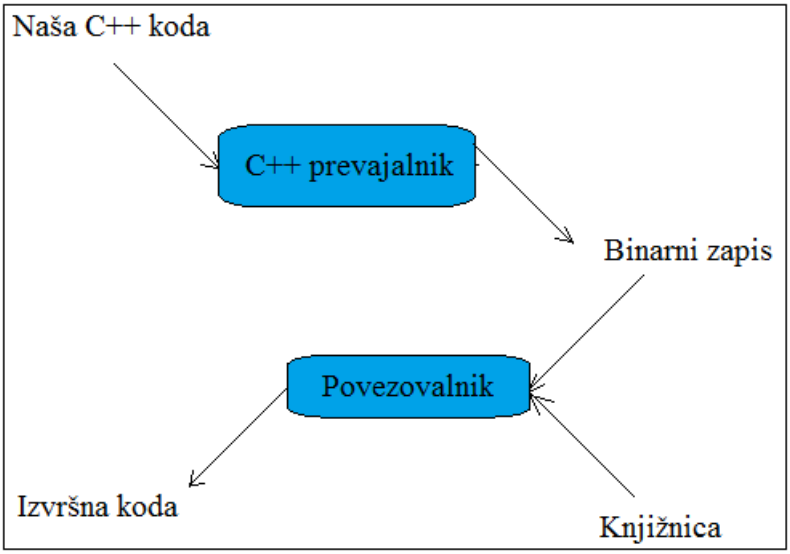
Keithley 2016p lahko upravljamo z računalnikom preko vodila RS-232 ali GPIB.

3 Programski jezik C++

Koncem sedemdestih let je Bjarne Stroustrup (takrat inženir pri Bell Labs) začel eksperimentirati z razširitvijo programskega jezika C. Ime C++ [8, 9, 10] se prvič pojavi leta 1983.

C++ je standardiziran, objektni, višji programski jezik (tretje generacije) katerega s prevajanjem programske kode pretvorimo v strojni jezik.

Slika 3.1 prikazuje postopek prevajanja programa. Prevajanje deluje tako, da prevajalnik (ang. compiler) najprej prevede celoten program v binarne datoteke (ang. object files), ki jih zatem povezovalnik (ang. linker) poveže v izvršni program (ang. executable). Prevajanje nam omogoča hitro delovanje naših programov, vendar je potrebno program prevesti za vsako platformo posebej.



Slika 3.1 : Prevajanje programa

V osnovi za razvijanje programske opreme v jeziku C++ ne potrebujemo ničesar drugega kot tekstovni urejevalnik za pisanje programske (izvorne) kode ter prevajalnik in povezovalnik. Razvojna okolja poleg tega ponujajo še napredni urejevalnik, razhroščevalnik (ang. debugger) ter sisteme za avtomatsko generiranje dokumentacije, različna načrtovalna orodja itn.. Pri

uporabi razvojnega okolja, je prevajanje oddaljeno le en klik ali pritisk na določeno funkcijsko tipko.

Program je zaporedje ukazov. Tako imenovana vstopna točka v programu pove operacijskemu sistemu, kje naj se program začne izvajati. V programskem jeziku C++ je ta vstopna točka funkcija **main** in je del vsakega C++ programa.

C++ podpira ločeno prevajanje, kar nam omogoča pisanje organizirane in pregledne programske kode. To v bistvu pomeni, da je celoten program predstavljen kot skupek ločenih datotek, katere so medseboj povezane. Za nas sta pomembni datoteki s končnico .h (ang. Header files-V nadaljevanju H) in .cpp (kot **c p**lus **p**lus; ang. Source files - v nadaljevanju CPP). V H datotekah deklariramo spremenljivke (določimo njihova imena in tip podatka) in vklapljamo potrebne programske knjižnice (ang. libraries).

Programska knjižnica je skupek že napisanih, pogosto uporabljanih funkcij, konstant, objektov, ki so na razpolago pri pisanju programa. V CPP datotekah definiramo spremenljivke, do katerih dostopamo z vključevanjem (ang. include) H datoteke, v katerem je ta spremenljivka deklarirana. CPP datoteke so tiste, ki se prevajaju in med sabo načeloma niso direktno povezane.

4 Programska abstrakcija inštrumentov

Preden se lotimo razlage programske kode C++ razredov (ang. Class), razjasnimo kaj je to abstrakcija [17]. To je proces razdelitve enega problema na več razvojnih nivojev oziroma plasti (ang. Layers). Vsak nivo se ukvarja le s tistimi podrobnostmi in vprašanji glede problema, ki so pomembne za trenutno perspektivo. Višji nivoji abstrakcije so sestavljeni iz nižjih nivojev, kar pomeni, da je njihova zanesljivost delovanja odvisna od zanesljivosti nižjih nivojev.

Primer takšne abstrakcije v računalništvu je programska oprema (ang. Software). Najvišji nivo v obliki aplikacij in operacijskih sistemov je sestavljen iz pravil nivoja programskega jezika. Nivo programskega jezika je odvisen od pravil nivoja strojne kode ter še nižje, pravil nivoja binarnih vezij. Abstrakcija torej omogoča analizo in sintezo zelo komplesnih sistemov.

Za dosego avtomatiziranega upravljanja izbranih inštrumentov smo problem razdelili na tri nivoja abstrakcije. Najnižji nivo je programiranje komunikacije med inštrumentom in računalnikom. Drugi nivo je programiranje vsake funkcije inštrumenta posebej ter njihovo povezovanje v skupni C++ razred. Naslednji je nivo razvoja naših demonstracijskih programov, kar je tudi končni nivo naše obravnave v diplomski nalogi. Ti programi so praktični primeri uporabe tretjega nivoja ker so sestavljeni iz kombinirane uporabe funkcij vseh treh inštrumentov, z namenom zajemanja želene karakteristike vezja. Na doseženem nivoju se od uporabnika, za osnovne primere uporabe zahteva le začetniška stopnja programiranja v programskem jeziku C++.

4.1 Komunikacija

Za smisleno delovanje programa je najprej potrebno ustvariti povezavo med inštrumentom in računalnikom. V našem primeru smo z inštrumenti komunicirali preko povezave RS−232, kjer je potrebno za vsak inštrument nastaviti odgovarjajočo številko serijskih vrat (ang. ComPort) ter hitrost prenosa podatkov (ang. Baudrate).

Koda komunikacije med inštrumentom in računalnikom je za vse inštrumente enaka. Za komunikacijo med računalnikom in povezanim inštrumentom skrbita H in CPP datoteki **communication** v knjižnici avtomatizacije tega inštrumenta. V teh datotekah deklariramo in definiramo spremenljivke za nastavljanje parametrov povezave, kot so številka serijskih vrat, hitrost prenosa, velikost medpomnilnika itn.. Prvi korak, programiranja komunikacije med inštrumenti, je odpiranje komunikacijskih vrat (ang. ComPort) s funkcijo operacijskega sistema Windows **CreateFileA**, kateri kot parameter podamo številko serijskih vrat ter ostale parametre, kot je prikazano v kodi na sliki 4.1.

```
01. RS232::RS232(unsigned int ComPort, unsigned int AReadBufferSize, char eol_mark) {
02.     if(ComPort > 99) { throw Ex_OpremaEA_Param(); }
03.
04.     char ComName[] = { 'C', 'O', 'M', '-', '\0', '\0' };
05.
06.     if(ComPort > 9) {
07.         ComName[3] = '0' + ComPort/10;
08.         ComName[4] = '0' + ComPort%10;
09.     } //if(ComPort > 9)
10.     else {
11.         ComName[3] = '0' + ComPort;
12.     } //else(ComPort < 9)
13.
14.     SerialComm = CreateFileA(ComName,
15.                              GENERIC_READ |GENERIC_WRITE,
16.                              0,
17.                              NULL,
18.                              OPEN_EXISTING,
19.                              0,
20.                              NULL); //FILE_FLAG_OVERLAPPED
21.
22. if (SerialComm == INVALID_HANDLE_VALUE){
23.     throw Ex_OpremaEA_Open();
24.     }   //Handle Error Condition
```

Slika 4.1 : Odpiranje komunikacijskih vrat

Najpomembnejši vidik programiranja serijske komunikacije je nastavitev strukture Kontrolni Blok Naprave (ang. Device Control Block – v nadaljevanju DCB), ker sicer povezava ne deluje pravilno. Za njeno inicijalizacijo smo uporabili funkcijo **GetCommState**. Ta vrne DCB strukturo, ki je trenutno v uporabi za izbrana komunikacijska vrata (slika 4.2). Spremenili smo le hitrost prenosa v odvisnosti od uporabljenega inštrumenta.

```
01.  void RS232::SetPort(unsigned int BaudRate) {
02.      DCB dcbConfig;
03.
04.      if(GetCommState(SerialComm, &dcbConfig)) {
05.          dcbConfig.BaudRate    = BaudRate;
06.          dcbConfig.ByteSize    = 8;
07.          dcbConfig.Parity      = NOPARITY;
08.          dcbConfig.StopBits    = ONESTOPBIT;
09.          dcbConfig.fDtrControl = DTR_CONTROL_DISABLE;
10.          dcbConfig.fRtsControl = RTS_CONTROL_DISABLE;
11.      }
12.      else {
13.          throw Ex_OpremaEA_Open();
14.      }
15.
16.      if(!SetCommState(SerialComm, &dcbConfig)) {
17.          throw Ex_OpremaEA_Open();
18.      }
19.  } //void RS232::SetPort(unsigned int BaudRate)
```

Slika 4.2 : Nastavitve serijskih vrat

Implementacija funkcij operacijskega sistema **WriteFile** (v nadaljevanju **Write**) in **ReadFile** (v nadaljevanju **Read**) je prikazana na sliki 4.3. Uporaba teh funkcij nam omogoča pošiljanje ukazov inštrumentu oziroma branje njegovih odgovorov.

```
01.  void RS232::Write(const char *DataSent) {
02.      size_t len = strlen(DataSent);
03.      if(!len) { return; }
04.
05.      DWORD dwNumberOfBytesWritten;
06.
07.      if(WriteFile(SerialComm, DataSent, len, &dwNumberOfBytesWritten, NULL) == 0) {
08.          throw Ex_OpremaEA_Write();
09.      }
10.      if(dwNumberOfBytesWritten != len) {
11.          throw Ex_OpremaEA_Write();
12.      }
13.
14.  }
15.
16.  size_t RS232::Read(char *ReadBuffer, size_t ReadBufferSize) {
17.
18.      DWORD dwNumberOfBytesRead;
19.
20.      if(ReadFile(SerialComm, ReadBuffer, ReadBufferSize, &dwNumberOfBytesRead, NULL) == 0) {
21.          throw Ex_OpremaEA_Read();
22.      }
23.
24.      if(dwNumberOfBytesRead != ReadBufferSize) {
25.      throw Ex_OpremaEA_Read();
26.      }
27.
28.      return dwNumberOfBytesRead;
29.  }
```

Slika 4.3 : Najnižji nivo komunikacije z inštrumentom

4.2 Razvoj C++ razredov

Vse programsko dostopne funkcije inštrumentov smo abstrahirali v C++ razrede, kar opisujemo na tem mestu; razlago smo omejili na najpogosteje uporabljene funkcije. Na sliki 4.4 je spisek standardnih C++ knjižnic, uporabljenih v razvoju naših knjižnic. Elementi teh knjižnic so definirani znotraj imenskega prostora (ang. Namespace) *std.*

```
01. #include <windows.h>
02. #include <iostream>
03. #include <fstream>
04. #include <conio.h>
05. #include <vector>
06. #include <stdio.h>
07. #include <iomanip>
08. #include <algorithm>
09. #include <numeric>
10. #include <math.h>
11.
12. using namespace std;
```

Slika 4.4 : Uporabljene standardne C++ knjižnice

4.2.1 Knjižnica inštrumenta Hameg HM8118

Oglejmo si ozadje razvitega C++ razreda. V datoteki **communication.h** smo deklarirali podatkovno strukturo RS232 v kateri so deklarirane in definirane spremenljivke zadolžene za komunikacijo. V deklaraciji podatkovne strukture Hameg (v datoteki Hameg.h, slika 4.5) smo uporabili ukaz dostopnosti (ang. Access specifier) **public** in podatkovno strukturo RS232. S tem je struktura RS232 (ter vsi njeni členi) postala dostopna znotraj strukture Hameg. Strukture so namenoma ločene, zaradi večje preglednosti in njunih območji zadolžitev.

V strukturi Hameg so deklarirane in definirane vse funkcije, objekti, razredi in spremenljivke, ki nam omogočajo dostop do nastavitvenih in nadzornih funkcij inštrumenta. Programiranje nastavitvenih funkcij inštrumenta smo razdelili v dve skupini. Če parameter funkcije nastavljamo oziroma pošiljamo inštrumentu, kličemo ustrezno funkcijo Set. V primeru, da želimo vrednost parametra prebrati z inštrumenta, kličemo ustrezno funkcijo Get.

```
01. struct Hameg: public RS232 {
02.
03.     void Set_ManualRange(int OffOn);
04.
05.     void Set_PMOD(SetupCommands Parametar);
06.     SetupCommands Get_PMOD();
07.
08.     void Set_SpeedRate(SetupCommands Speed);
09.     SetupCommands Get_SpeedRate();
10.
11.     unsigned int OperationComplete();
12.
13.     char* EnumToString(SetupCommands x);
14.
15.     void Set_Frequency(unsigned int Frekvenca);
16.     int Get_Frequency();
17.
18.     void Set_Volt(double Napetost);
19.     double Get_Volt();
20.
21.     double MainResult();
22.
23.     double SecondResult();
24. }
```

Slika 4.5 : Del strukture Hameg

Slika 4.5 prikazuje del strukture Hameg, v katerem so deklarirane funkcije, katerih postopek programiranja smo razložili v nadaljevanju poglavja. Prva deklarirana funkcija v strukturi Hameg, **Set_ManualRange** nam omogoča vklop in izklop ročnega nastavljanja prikaza območja rezultata za izbrani parameter. V našem primeru smo ročno nastavljanje izklopili. S tem smo izkoristili možnost inštrumenta, da sam določi območje prikaza parametra. Inštrument samodejno izbere optimalno območje prikaza glede na vrednost rezultata meritve. Če se rezultat zniža pod 22,5 % obsega trenutno nastavljenega območja, inštrument prestavi na nižje območje prikaza. V primeru, ko rezultat preseže 90 % obsega, inštrument prestavi na višje območje prikaza.

Naslednji deklarirani parameter (SetupCommands Parameter) funkcije **Set_PMOD** je tisti, ki ga določi uporabnik. Ker računalnik z inštrumenti komunicira izključno z numeričnimi vrednostmi, smo uporabili strukturo enumerator. Njegova naloga je številčno vrednost povezati z besednim identifikatorjem [10] (Slika 4.6). To pomeni, da beseda ZTheta, na sliki 4.6, dejansko pomeni parameter številka 6. S tem smo uporabniku, namesto določanja številke parametra, izbiro povezali s človeku intuitivnejšo oznako tega parametra.

```
01. static enum SetupCommands { AUTO,
02.                             LQ,
03.                             LR,
04.                             CD,
05.                             CR,
06.                             RQ,
07.                             ZTheta,
08.                             YTheta,
09.                             RX,
10.                             GB,
11.                             NTheta,
12.                             M,
13.                             OFF,
14.                             ON,
15.                             FAST,
16.                             MED,
17.                             SLOW,
18.                             CONT,
19.                             MAN,
20.                             EXTERNAL,
21.                             INTERNAL,
22.                             SER,
23.                             PAR,
24.                             NORMAL,
25.                             ABS,
26.                             REF_M,
27.                             REF_S};
```

Slika 4.6 : Deklaracija enumeratora

Funkcijo **EnumToString** (kar v prevodu pomeni številka-v-besedo, Slika 4.7) uporabljamo, ko želimo izvedeti trenutno nastavitev parametra na inštrumentu. Njena funkcija je, da dobljeno vrednost parametra primerja z vrednostmi **switch-case** zanke [10] ter vrne odgovarjajočo oznako parametra.

Funkcijo **Set_SpeedRate** smo izbrali za prikaz uporabe strukture enumerator. S funkcijo **Set_SpeedRate** nastavljamo hitrost izvajanja meritev. Inštrument ima možnost nastavljanja hitrosti na HITRO (ang. FAST), SREDNJE (ang. MEDIUM) ali POČASNO (ang. SLOW), kar se zakodira z vrednostmi 0, 1 ali 2. Torej je potrebno inštrumentu poslati ukaz v obliki ‘RATE2’. Ker vrednosti 0, 1 in 2 niso intuitivne uporabimo strukturo enumerator [10].

```
01.  char* Hameg::EnumToString(SetupCommands x) {
02.
03.      switch (x) {
04.          case Hameg::AUTO:
05.              return "AUTO";
06.              break;
07.          case Hameg::LQ:
08.              return "LQ";
09.              break;
10.          case Hameg::LR:
11.              return "LR";
12.              break;
13.          case Hameg::CD:
14.              return "CD";
15.              break;
16.          case Hameg::CR:
17.              return "CR";
18.              break;
19.          case Hameg::RQ:
20.              return "RQ";
21.              break;
22.          case Hameg::ZTheta:
23.              return "ZTheta";
24.              break;
25.          case Hameg::YTheta:
26.              return "YTheta";
27.              break;
28.          case Hameg::RX:
29.              return "RX";
30.              break;
31.          case Hameg::GB:
32.              return "GB";
33.              break;
34.          case Hameg::NTheta:
35.              return "NTheta";
36.              break;
37.          case Hameg::M:
38.              return "M";
39.              break;
40.                .
41.                .
42.                .
```

Slika 4.7 : Začetni del definicije funkcije EnumToString (elementi merilnih funkcij)

Na sliki 4.8 je prikazan klic funkcije **Set_SpeedRate** v datoteki **Main.cpp**. Oznako ‘y’ smo uporabili kot kazalec (ang. Pointer) [9] na strukturo Hameg. Za klic katerekoli funkcije, definirane v strukturi Hameg, je potrebno uporabiti kazalec ‘y’. Številka 3 so komunikacijska vrata preko katerih poteka komunikacija z inštrumentom. Parameter funkcije **Set_SpeedRate** je element strukture enumerator.

```
01.  void main() {
02.
03.      Hameg y(3);
04.
05.      y.Set_SpeedRate(y.MED);  // Measuring speed FAST,MED,SLOW
06.
07.      }
```

Slika 4.8 : Klic funkcije Set_SpeedRate

Element MED (skrajšano MEDIUM) je 15. element strukture enumerator, torej je njegova vrednost 15. Njegova sosednja elementa v strukturi sta FAST in SLOW, ki imata vrednosti 14 in 16. Za nastavljanje SREDNJE hitrosti je potrebno inštrumentu poslati ukaz 'RATE1'. V šesti vrstici na sliki 4.9 od vrednosti, ki jo ima želeni parameter, odštejemo 14 in novo vrednost shranimo v spremenljivko a. Če je vrednost spremenljivke a različna od 0, 1 ali 2, pomeni, da je uporabnik izbral napačen element strukture enumerator.

```
01. #include "Hameg.h"
02.
03.
04. void Hameg::Set_SpeedRate(SetupCommands Speed) {
05.
06.     int a = (int)Speed - 14;
07.     if (a<0 || a>2){
08.         throw Ex_OpremaEA_Param();
09.     }
10.
11.     WriteBuffer = new char[WriteBufferSize];
12.
13.     WriteBuffer[0] = 'R';
14.     WriteBuffer[1] = 'A';
15.     WriteBuffer[2] = 'T';
16.     WriteBuffer[3] = 'E';
17.
18.     WriteBuffer[4] = '0' + a;
19.
20.     WriteBuffer[5] = '\r';
21.     WriteBuffer[6] = '\0';
22.
23.     Write(WriteBuffer);
24. }
```

Slika 4.9 : Definicija funkcije Set_SpeedRate

Če uporabnika zanima povratna informacija oziroma nastavljena hitrost inštrumenta, uporabi funkcijo **Get_SpeedRate** (Slika 4.10).

```
01. Hameg::SetupCommands Hameg::Get_SpeedRate(){
02.     Write("RATE?\r");
03.     size_t len = Read(ReadBuffer, ReadBufferSize);
04.     ReadBuffer[len] = '\0';
05.     if(len > 3) {
06.         throw Ex_OpremaEA_Logic();
07.                 }
08.     int Mode = atoi (ReadBuffer);
09.     Mode = Mode + 14;
10.     SetupCommands PM = (SetupCommands) Mode;
11.     return PM;
12. }
```

Slika 4.10 : Definicija funkcije Get_SpeedRate

Vrnjeni vrednosti funkcije **Get_SpeedRate** (0,1 ali 2) zdaj prištejemo vrednost 14 (9. vrstica) ter poiščemo element strukture enumerator, ki se nahaja na mestu dobljene vrednosti. Prej smo

omenili, da je naloga funkcije **EnumToString** ta element prikazati uporabniku. Kot vidimo na sliki 4.11, funkcija vrne niz znakovnih elementov za dobljeni element strukture enumerator.

```
01.         .
02.         .
03.         .
04.  case Hameg::FAST:
05.      return "FAST";
06.      break;
07.  case Hameg::MED:
08.      return "MED";
09.      break;
10.  case Hameg::SLOW:
11.      return "SLOW";
12.      break;
13.         .
14.         .
15.         .
```

Slika 4.11 : Del definicije funkcije EnumToString (elementi hitrosti izvajanja meritev)

Slika 4.12 prikazuje kodo praktične uporabe funkcije **Get_SpeedRate**. V 8. vrstici je klic funkcije **EnumToString**, ki nam vrne parameter hitrosti izvajanja meritev v obliki, primerni za prikaz na zaslonu.

```
01.  void main() {
02.
03.      Hameg::SetupCommands s; //SPEED RATE
04.      Hameg y(3);
05.
06.      s = y.Get_SpeedRate();
07.
08.      cout << "Speed Rate: " << y.EnumToString(s) << '\n';  // Measuring speed FAST,MED,SLOW
09.
10.      }
```

Slika 4.12 : Klic funkcije Get_SpeedRate

Hitrost prenosa podatkov in ukazov med računalnikom in inštrumentom je končna zaradi končne hitrosti odziva inštrumenta. Nadzorno funkcijo **OperationComplete** uporabljamo, ko se želimo prepričati, da je inštrument končal s predhodno aktiviranimi meritvami in je pripravljen na novi ukaz. V primeru vzporednega izvajanja večjega števila različnih meritev, obstaja nevarnost napačnih rezultatov. Uporaba funkcije **OperationComplete** poskrbi, da inštrument ne začne z izvajanjem naslednje meritve, preden ne zaključi s prejšnjo.

Funkcija **Set_Volt** (slika 4.13) nam omogoča nastavljanje efektivne vrednosti generirane sinusne napetosti. Parameter funkcije **Set_Volt** je spremenljivka Napetost, podatkovnega tipa

double. Pri pošiljanju ukazov, ki imajo spremenljiv parameter, je potrebno vnaprej določiti maksimalno dolžino ukaza. S tem določimo tudi mesti v medpomnilniku pisanja (WriteBuffer), kjer je potrebno dodati znaka za konec ukaza '\r' in '\0'. Spremenljivka Napetost je tipa double, torej je inštrumentu možno poslati vrednost parametra v decimalni obliki. Omejitve inštrumenta zahtevajo vrednost parametra med 0,05 V in 1,5 V. Vrednosti z več kot dvema decimalkama se s korakom 0,01 V zaokrožijo navzgor. Za boljše razumevanje programske kode smo v nadaljevanju vzeli poljubno vrednost (recimo 1,372334) in sledili njenemu zapisu do končnega ukaza.

```
01. #include "Hameg.h"
02.
03. void Hameg::Set_Volt(double Napetost) {
04.
05.     Napetost = Napetost*1000;
06.     int Nap = (int)Napetost;
07.     if (Nap < 5 || Nap > 1500){
08.         throw Ex_OpremaEA_Param();
09.     }
10.
11.     WriteBuffer = new char[WriteBufferSize];
12.
13.     WriteBuffer[0] = 'V';
14.     WriteBuffer[1] = 'O';
15.     WriteBuffer[2] = 'L';
16.     WriteBuffer[3] = 'T';
17.
18.     WriteBuffer[4] = '0' + Nap/1000;
19.
20.     Nap = Nap%1000;
21.
22.     WriteBuffer[5] = '.';
23.     WriteBuffer[6] = '0' + Nap/100;
24.
25.     Nap = Nap%100;
26.
27.     int Prenos = Nap%10;
28.
29.     if (Prenos != 0){
30.         Nap = Nap+10;
31.     }
32.
33.     WriteBuffer[7] = '0' + Nap/10;
34.
35.     WriteBuffer[8] = '\r';
36.     WriteBuffer[9] = '\0';
37.
38.     Write(WriteBuffer);
39. }
```

Slika 4.13 : Definicija funkcije Set_Volt

Na začetku vrednost parametra, zapisano s strani uporabnika, pomnožimo s 1000 (1372,334) in jo shranimo kot vrednost spremenljivke Nap, podatkovnega tipa int (1372). Nato preverimo, če je ta vrednost znotraj omejitev inštrumenta.

V 11. vrstici (slika 4.13) smo medpomnilniku WriteBuffer dodelili količino pomnilnika enake vrednosti spremenljivke WriteBufferSize. S tem si zagotovimo prostor za vpis vrednosti parametra. Na prva štiri mesta medpomnilnika vpišemo črke 'V' 'O' 'L' 'T', kar je nespremenljivi del ukaza. Na peto mesto vpišemo vrednost celoštevilčnega deljenja vrednosti spremenljivke Nap s 1000 (1372 / 1000 = 1).

Kot novo vrednost spremenljivke Nap zapišemo vrednost ostanka deljenja Nap s 1000 (1372 % 1000 = 372). Zdaj vemo, da v zapisu sledi pika, ker je vrednost pred decimalno piko v parametru lahko le 0 ali 1. Vrednost Nap (372) zdaj celoštevilčno delimo s 100 in rezultat vpišemo na sedmo mesto medpomnilnika (372 / 100 = 3). Ostanek deljenja Nap s 100 shranimo nazaj v Nap (372 % 100 = 72). Ker je korak zokrožitve 0,01 navzgor, je potrebno preveriti, če je ostanek deljenja Nap z 10 večje od 0 (72 % 10 = 2). V primeru, ko je ostanek res različen od 0, vrednosti Nap prištejemo 10 (72 + 10 = 82). Na osmo mesto medpomnilnika vpišemo vrednost celoštevilčnega deljenja Nap z 10 (82 / 10 = 8). Vpišemo še znaka za konec ukaza ter ukaz pošljemo inštrumentu s funkcijo **Write**. Končno stanje medpomnilnika v prikazanem primeru je 'VOLT1.38'.

Za branje rezultatov uporabljamo funkciji **MainResult** in **SecondResult**. Inštrument meri in prikazuje dva parametra hrati, zato želimo imeti možnost branja rezultatov obeh. Funkcija **MainResult** prebere in vrne vrednost prvega, funkcija **SecondResult** pa vrednost drugega parametra. Slika 4.14 prikazuje potek komunikacije med računalnikom in inštrumentom pri branju rezultata parametrov.

```
01.  #include "Hameg.h"
02.
03.  double Hameg::MainResult() {
04.      Write("XMAJ?\r");
05.      size_t len = Read(ReadBuffer, ReadBufferSize);
06.      ReadBuffer[len] = '\0';
07.              //if(len > 3) { throw Ex_OpremaEA_Logic(); }
08.
09.      double MainResult = atof (ReadBuffer);
10.      return MainResult;
11.  }
12.
13.  double Hameg::SecondResult() {
14.      Write("XMIN?\r");
15.      size_t len = Read(ReadBuffer, ReadBufferSize);
16.      ReadBuffer[len] = '\0';
17.              //if(len > 3) { throw Ex_OpremaEA_Logic(); }
18.
19.      double SecondResult = atof (ReadBuffer);
20.      return SecondResult;
21.  }
```

Slika 4.14 : Definiciji funkcij MainResult in SecondResult

V četrti vrstici s funkcijo **Write** pošljemo ukaz, zapisan vedno v enaki obliki in brez dodatnih parametrov, vendar z znakom '\r', ki ukaz zaključi. Peta vrstica prebere odgovor inštrumenta ter dolžino rezultata zapiše v spremenljivko len podatkovnega tipa size_t.

Za odgovor inštrumenta smo pripravili količino pomnilnika, določeno s spremenljivko ReadBufferSize. Rezultat dobi smisleno obliko šele, ko vstavimo znak za konec znakovnega niza ('\0') na mesto v medpomnilniku branja (ReadBuffer), ki je enako vrednosti spremeljivke len. V šesti vrstici smo vrednost v medpomnilniku ReadBuffer, podatkovnega tipa char, s funkcijo **atof** spremenili v vrednost podatkovnega tipa double. Ta pretvorba nam omogoča uporabo rezultata v nadaljevanju programa. Funkcija **atof** je del standardne C++ knjižnice *stdlib.h*.

4.2.2 Knjižnica inštrumenta Keithley 2016p

V izvorni datoteki **Keithley.cpp** in H datoteki **Keithley.h** so deklarirane in definirane vse funkcije, objekti, razredi in spremenljivke, ki nam omogočajo dostop do nastavitvenih in nadzornih funkcij inštrumenta Keithley 2016p. Deklaracije in definicije funkcij, ki so pomembne za našo nalogo, smo prikazali na sliki 4.15. Glede na njihovo delovanje in namen smo jih razvrstili v štiri skupine in vsako skupino posebej podrobneje razložili.

```
01. struct Keithley: public RS232 {
02.
03.     void ResetInstrument();
04.     void ClearStatus();
05.     const char* Get_OperationComplete();
06.     void Set_ContinuousInitiationState(unsigned int Boolean);
07.     void Set_OperationLoopsNumInTrigger(unsigned int Count);
08.     void OneTriggerCycle();
09.     void Set_MeasurementFunction(Functions Mode);
10.     void Set_DistInputSignalFreq(unsigned int Frequency);
11.
12.     void Set_SineWave50OhmOutput(unsigned int Frequency,double Amplitude);
13.     //void Set_SineWave600OhmOutput(unsigned int Frequency,double Amplitude);
14.     //void Set_SineWaveHighImpedanceOutput(unsigned int Frequency,double Amplitude);
15.     double Get_SineWaveOutputFreq();
16.     double Get_SineWaveOutputAmpl();
17.     void Set_OutputState(unsigned int Boolean);
18. }
```

Slika 4.15 : Deklaracije uporabljenih funkcij iz Keithley.h

Splošni ukazi (ang. Common commands) so nadzorni ukazi. Uporabljamo jih za regulacijo začetnih nastavitev inštrumenta in za nadzorovanje poteka izvajanja meritev. Funkcija **ResetIntrument** prekliče izvajanje vseh ukazov in vrne inštrument v standardno začetno

stanje. S to funkcijo si zagotovimo, da je inštrument vedno enako nastavljen in da delovanje programa ni pogojeno s predhodnimi nastavitvami na samem inštrumentu.

Funkcija **ClearStatus** izbriše bite v registrih (nastavi na 0). Uporaba ene od teh funkcij (ali obeh) pošlje inštrument v stanje nedejavnosti (ang. Operation Complete Command Query Idle State-V nadaljevanju OQIS) pred začetkom izvajanja meritev. Za inštrument velja, da je v stanju nedejavnosti takrat, ko ne opravlja nobene operacije merjenja ali skeniranja.

Programiranje nadzornih funkcij inštrumenta Keithley poteka enako kot pri inštrumentu Hameg. Za hitro ponovitev postopka smo opisali programsko kodo funkcije **IdentifyInstrument** na sliki 4.16, ki vrne polno ime inštrumenta. S funkcijo **Write** pošljemo inštrumentu ukaz "*IDN?\r". Odgovor inštrumenta, ki je shranjen v medpomnilniku Readbuffer zaključimo z znakom '\0'. Za prikaz imena inštrumenta na zaslonu uporabimo predmet (ang. Object) **cout** iz standardne C++ knjižnice *ostream.h*.

```
01. const char* Keithley::IdentifyInstrument() {
02.     Write("*IDN?\r");
03.     size_t len = Read(ReadBuffer, ReadBufferSize);
04.     ReadBuffer[len] = '\0';
05.             //if(len > 3) { throw Ex_OpremaEA_Logic(); }
06.
07.     cout << "Instrument ID: " << ReadBuffer << '\n';
08.
09.     return ReadBuffer;
10. }
```

Slika 4.16 : Definicija funkcije IdentifyInstrument

Prožilne ukaze (ang. Trigger commands) uporabljamo za nastavljanje prožilnega modela izvajanja meritev. Funkciji **OneTriggerCycle** in **Set_ContinuousInitiationState** pošljeta prožilna ukaza, ki prestavita inštrument iz stanja nedejavnosti v aktivno stanje (ang. Operation Complete Command Query Active State-V nadaljevanju OQAS). Pri uporabi funkcije **OneTriggerCycle** se inštrument po enem krogu meritev vrne v OQIS. Za aktivirano funkcijo **Set_ContinuousInitiationState** velja, da inštrument sproža meritve do ukaza uporabnika za prekinitev. Funkcija **Get_OperationComplete** aktivira zastavico Ni-Čakajočega-Ukaza (ang. No-Operation-Pending flag). Zastavica dobi vrednost 1, ko se inštrument po izvajanju prožilnih ukazov vrne v OQIS. Inštrument takrat pošlje na izhod ASCII znak '1' [11] in program lahko nadaljuje z izvajanjem naslednjega ukaza.

Set_OperationLoopsNumInTrigger je funkcija s katero določamo število operacij, ki naj se izvedejo znotraj sprožene meritve.

Ukaze Zaznavanja (ang. Sense commands) uporabljamo za nastavljanje in nadzor meritvenih funkcij. Parameter funkcije **Set_MeasurementFunction** določa merilno funkcijo. Uporabnik ima na izbiro merilne funkcije prikazane na sliki 4.17. Določanje merilne funkcije smo poenostavili z uporabo strukture enumerator (poglavje 4.2.1).

```
01. static enum Functions { CurrentAC,
02.                         CurrentDC,
03.                         VoltageAC,
04.                         VoltageDC,
05.                         Resistance2,
06.                         Resistance4,
07.                         Period,
08.                         Frequency,
09.                         Temperature,
10.                         Diode,
11.                         Continuity,
12.                         Distortion
13. }
```

Slika 4.17 : Deklaracija enumeratora z merilnimi funkcijami

Inštrument Keithley 2016p ima na zadnji strani sinusni vir z nastavljivo amplitudo in frekvenco. Ukazi Izhoda (ang. Output commands) se uporabljajo za nadzor tega vira. Z njimi nastavljamo parametre izhodnega signala, njegovo frekvenco, amplitudo, impedanco in obliko (ang. Waveform). Funkcija **Set_OutputState** omogoča vklop in izklop generatorja izhodnega vira. Pred vključitvijo vira je potrebno nastaviti vse ostale parametre izhodnega signala. Uporabnik ima na izbiro tri funkcije, s katerimi nastavlja vrednost izhodne impedance:

- **Set_SineWave50OhmOutput** s 50 ohmsko,
- **Set_SineWave600OhmOutput** s 600 ohmsko ter
- **Set_SineWaveHighImpedanceOutput** s visoko ohmsko izhodno imedanco.

Parametra vseh funkcij sta frekvenca in amplituda, kar nam omogoča nastavljanje vseh parametrov generiranega izhodnega signala z eno samo funkcijo. Za branje nastavljenih vrednosti frekvence in amplitude generiranega izhodnega signala uporabljamo funkciji **Get_SineWaveOutputFreq** in **Get_SineWaveOutputAmpl**.

Programiranje nastavitvenih funkcij inštrumenta Keithley je bolj zapleteno. Razdeljene so v podsisteme, kot so računski podsistem, podsistemi zaslona, sprožanja, izhoda ter številni drugi. Vsak poslani ukaz inštrumentu (s funkcijo **Write**) mora poleg funkcije in parametra vsebovati še ime podsistema. Vpisovanje imena podsistemov v ukaz smo ločili od programiranja njihovih funkcij.

Na primeru funkcije **Get_OneShotMeasurement** smo razložili proces pisanja celotnega ukaza pred pošiljanjem. Funkcija **Get_OneShotMeasurement** nastavi merilno funkcijo, sproži meritev in vrne njen rezultat. Pri programiranju smo si ponovno pomagali z uporabo strukture enumerator.

```
01. double Keithley::Get_OneShotMeasurement(Functions Mode) {
02. 
03.     int a = (int)Mode; a = a++;
04.     if (a < 1 || a > 13){throw Ex_OpremaEA_Param();}
05. 
06.     WriteBuffer = new char[WriteBufferSize];
07.     WriteBuffer[0] = ':';
08.     WriteBuffer[1] = 'M';
09.     WriteBuffer[2] = 'E';
10.     WriteBuffer[3] = 'A';
11.     WriteBuffer[4] = 'S';
12.     WriteBuffer[5] = ':';
13. 
14.     Size = 6;
15. 
16.     int Measure[12] = {1,2,3,4,5,6,7,8,9,10,11,12};
17.     std::string ProgramMode[12] = {"CURR:AC", "CURR:DC", "VOLT:AC", "VOLT:DC",
18.                                    "RES", "FRES", "PER", "FREQ",
19.                                    "TEMP", "DIOD", "CONT", "DIST"};
20.     bool found = false;
21.     for(size_t i = 0; i < ProgramMode[i].max_size(); i++){
22.                 if(Measure[i] == a){
23.                     for(size_t j = 0; j < ProgramMode[i].length(); j++){
24.                         char c = ProgramMode[i][j];
25.                         WriteBuffer[6 + j] = c;
26.                         Size = Size++;
27.                     }
28.                     found = true;
29.                     break;
30.                 }
31.     }
32.     if(!found){throw Ex_OpremaEA_Param();}
33. 
34.     WriteBuffer[Size] = '?';
35.     WriteBuffer[1+Size] = '\r';
36.     WriteBuffer[2+Size] = '\0';
37.     Write(WriteBuffer);
38. 
39.     size_t len = Read(ReadBuffer, ReadBufferSize);
40.     ReadBuffer[len] = '\0';
41. 
42.     double Conf = atof (ReadBuffer);
43.     return Conf;
44. }
```

Slika 4.18 : Definicija funkcije Get_OneShotMeasurement

Prvi del kode na sliki 4.18 smo že srečali v prejšnjih poglavjih. Razlago smo zato začeli pri 16. vrstici programske kode, kjer smo inicializirali dva različna 12 členska polja (ang. Array), celoštevilsko in besedno. Celoštevilsko polje je sestavljeno iz številk od 1 od 12. Besedno polje je sestavljeno iz imena vseh merilnih funkcij inštrumenta, napisanih v posebni obliki.

Inštrumentu Hameg (poglavje 4.2.1) je pri nastavljanju merilnega parametra bilo potrebno poslati številčno vrednost elementa strukture enumerator, ki označuje ta parameter. Pri inštrumentu Keithley, je to številčno vrednost elementa potrebno spremeniti v določeni znakovni niz oziroma ukaz.

V **for** zanki [10] (21. vrstica) najprej primerjamo številčno vrednost elementa strukture z vrednostimi celoštevilskega polja. Za vrednost spremenljivke i, pri kateri sta ti dve vrednosti enaki, poiščemo i-ti člen besednega polja. Naloga druge **for** zanke (vrstica 23) je vpisovanje znakovnega niza, ki označuje iskani parameter, v medpomnilnik WriteBuffer. Na konec ukaza dodamo še niz » ?\r\0 «. Ukaz pošljemo inštrumentu s funkcijo **Write** ter njegov odgovor spremenimo v podatkovni tip double.

Razložimo še funkciji **Set_TextASCIImessage** in **Get_TextASCIImessage** iz podsistema zaslona. Razlaga njune kode je pomembna, kljub dejstvu, da njuna uporaba ni pogosta v procesu izvajanja meritev. Funkciji prikažeta tekstovno sporočilo na zaslonu inštrumenta.

```
01. const char* Keithley::Get_TextASCIImessage(char* Text){
02.     Write(":DISP:ENAB 1;:DISP:TEXT:STAT 1\r");
03.
04.     Set_TextASCIImessage(Text);
05.
06.     Write(":DISP:TEXT:DATA?\r");
07.     size_t len = Read(ReadBuffer, ReadBufferSize);
08.     ReadBuffer[len] = '\0';
09.             //if(len > 3) { throw Ex_OpremaEA_Logic(); }
10.
11.     cout << "TextASCIImessage: " << ReadBuffer << '\n';
12.
13.     return ReadBuffer;
14. }
```

Slika 4.19 : Definicija funkcije Get_TextASCIImessage

S funkcijo **Write** (2. vrstica na sliki 4.19) pošljemo ukaza ‘:DISP:ENAB 1’ in ‘:DISP:TEXT:STAT 1’, s katerima vklopimo funkciji zaslona inštrumenta in tekstovnega sporočila. V funkciji **Set_TextASCIImessage** (Slika 4.20) sporočilo oziroma parameter Text

spremenimo v podatkovni tip string. S tem smo poskrbeli, da je sporočilo TextS možno obravnavati kot polje znakov.

V **for** zanki (slika 4.20) smo omejili število znakov v sporočilu na 12. V primeru, da je sporočilo krajše od 12 znakov, **if** zanka [10] pošlje signal za konec izvajanja **for** zanke. Po končanem vpisovanju sporočila dodamo še znak '**"**'. V 8. vrstici smo začeli vpisovati znake v medpomnilnik WriteBuffer. Prvi znak smo zapisali na 12. mestu v medpomnilniku. Kot je že omenjeno, smo oznake podsistemov ločili od funkcij. Na prvih 11. mest medpomnilnika WriteBuffer vpišemo oznako podsistema v kodi na sliki 4.21.

```
01.  void Keithley::Set_TextASCIImessage(char* Text){
02.
03.      std::string TextS;
04.      TextS = Text;
05.
06.      WriteBuffer = new char[WriteBufferSize];
07.      WriteBuffer[11] = 'D';
08.      WriteBuffer[12] = 'A';
09.      WriteBuffer[13] = 'T';
10.      WriteBuffer[14] = 'A';
11.      WriteBuffer[15] = ' ';
12.      WriteBuffer[16] = '"';
13.
14.      Size = 12;
15.      for(int i = 0; i < 12; i++){
16.          char c = TextS[i];
17.          if (c == 0)
18.              i = 12;
19.          else {
20.          WriteBuffer[17 + i] = c;
21.          Size = Size++;
22.          }
23.      }
24.      int Txt = Size - 12;
25.      WriteBuffer[17+Txt] = '"';
26.
27.      _TextDisplay();
28.      _Display();
29.  }
```

Slika 4.20 : Definicija funkcije Set_TextASCIImessage

Pomembno vlogo pri programiranju številnih funkcij inštrumenta Keithley smo dodelili spremenljivki Size. Njena vrednost določa mesto v medpomnilniku WriteBuffer, kjer je potrebno dodati znaka '\r' in '\0'.

```
01. void Keithley::_Display(){
02.     WriteBuffer[0] = ':';
03.     WriteBuffer[1] = 'D';
04.     WriteBuffer[2] = 'I';
05.     WriteBuffer[3] = 'S';
06.     WriteBuffer[4] = 'P';
07.     WriteBuffer[5] = ':';
08.
09.     WriteBuffer[6 + Size] = '\r';
10.     WriteBuffer[7 + Size] = '\0';
11.     Write(WriteBuffer);
12. }
13. void Keithley::_TextDisplay(){
14.     WriteBuffer[6] = 'T';
15.     WriteBuffer[7] = 'E';
16.     WriteBuffer[8] = 'X';
17.     WriteBuffer[9] = 'T';
18.     WriteBuffer[10] = ':';
19. }
```

Slika 4.21 : Vpisovanje oznake podsistema zaslona

4.2.3 Knjižnica napajalnika TTi QL355TP

V H datoteki **PowerSupply.h** so deklarirane vse funkcije inštrumenta, ki so nam na razpolago tudi pri ročnem nastavljanju. Deklaracije funkcij napajalnika, ki so pomembne za našo nalogo, smo prikazali na sliki 4.22. Njihovo delovanje ter posebnosti v kodi njihove definicije smo razložili v nadaljevanju poglavja.

```
01. struct QL355TP: public USB_RS232 {
02.
03.     void ResetInstrument();
04.     void ClearStatus();
05.     unsigned int OperationComplete();
06.
07.     void Set_AllOnOff(int OnOff);
08.     void Set_OnOff(int Output, int OnOff);
09.
10.     void Set_OutputV(int Output, double Volts);
11.     double Get_OutputV(int Output);
12.     void Set_StepSizeVoltage(int Output, double Volts);
13.     void IncVStepSize(int Output);
14.     void IncVerifyStepSizeV(int Output);
15. }
```

Slika 4.22 : Deklaracija uporabljenih funkcij iz PowerSupply.h

Nadzorna funkcija **ResetInstrument** ponastavi inštrument na tovarniško privzete nastavitve. Funkcija **ClearStatus** počisti status registrov pred izvajanjem novih meritev.

Nastavitveni funkciji inštrumenta, **Set_AllOnOff** in **Set_OnOff**, uporabljamo pri aktivaciji izhodov inštrumenta. Za aktivacijo/deaktivacijo določenega izhoda uporabimo funkcijo **Set_OnOff**. Prvi parameter funkcije je oznaka oziroma številka izhoda, z drugim pa določimo

njegov status (aktiviran/deaktiviran). Funkcija **Set_AllOnOff** aktivira/deaktivira vse izhode inštrumenta naenkrat, potrebuje samo parameter statusa.

Nastavitvene funkcije izhodov uporabljamo za nastavitev napetosti in toka na izhodih. Prvi parameter teh funkcij je številka izhoda, na katerem spreminjamo vrednosti. Delovanje in uporaba teh funkcij je možna le v primeru, ko je izbrani izhod deaktiviran. S funkcijo **Set_OutputV** nastavljamo želeno napetost na izhodu. Najvišja nastavljiva vrednost je 15 V ali 35 V, odvisno od izbranega območja toka in napetosti (poglavje 2.2). Uporaba funkcije **Get_OutputV** vrne trenutno nastavljeno vrednost napetosti na izhodu.

Korak spreminjanja (ang. Step size) vrednosti napetosti nastavljamo s funkcijo **Set_StepSizeVoltage**. Vrednost napetosti na izhodu spremenimo za vrednost nastavljenega koraka z uporabo funkcij **IncVStepSize** in **DecVStepSize** (ang. Increment = prišteje in ang. Decrement = odšteje). Pri uporabi funkcije **IncVerifyStepSizeV** inštrument, zraven prištevanja vrednosti koraka preveri, ali je nova vrednost napetosti znotraj omejitev nastavljenega območja napetosti.

Pri programiranju funkcij napajalnika je potrebno na konec ukaza (poleg znaka '\r') dodati znak '\n', ki označuje skok v naslednjo vrstico. Torej je za poizvedbo imena inštrumenta potrebno poslati ukaz "*IDN?\r\n".

Napajalnik ima tri izhode, zato je pri programiranju večine nastavitvenih funkcij potrebno (kot parameter) določiti, na katerem izhodu spreminjamo/nastavljamo veličino in njeno vrednost. Kot primera smo razložili programsko kodo funkcije **Set_OutputV** in kombinirane funkcije **MaxV**.

Slika 4.23 prikazuje kodo funkcije **Set_OutputV**. Z njo nastavljamo vrednost napetosti na izbranem izhodu. V deklaraciji smo določili, da funkcija potrebuje dva parametra. Prvi parameter je izbira izhoda, ki lahko zavzame vrednosti 1, 2 ali 3. Drugi parameter je želena vrednost napetosti. Vrednost napetosti izbiramo med 0 in maksimalno možno napetostjo, ki jo določa nastavljeno območje.

Naslednji korak programiranja je pisanje ukaza v medpomnilnik WriteBuffer. Najprej vpišemo oznako veličine, ki jo nastavljamo, in vrednost izhoda, na katerem to počnemo. Maksimalna napetost je 35 V. Vse manjše vrednosti je možno pošiljati na tri decimale natančno. Torej je največje število znakov, ki jih zasede poslana vrednost napetosti enako 5. Poslane vrednosti, manjše od 10 V, so sestavljene iz največ štirih znakov, vrednosti večje od 10 V pa iz pet. Programska koda se zato razlikuje med obema primeroma. Postopek vpisovanja vrednosti je podoben kot v primeru funkcije Set_Volt (inštrument Hameg HM8118), ki smo ga razložili v poglavju 4.2.1.

```
01.  void QL355TP::Set_OutputV(int Output, double Volts){
02.      if(Output < 0 || Output > 3) {
03.          throw Ex_OpremaEA_Param();
04.      }
05.      if (Volts < 0 || Volts > MaxV(Output)) {
06.          throw Ex_OpremaEA_Param();
07.      }
08.
09.      WriteBuffer = new char[WriteBufferSize];
10.      WriteBuffer[0] = 'V';
11.      WriteBuffer[1] = '0' + Output;
12.      WriteBuffer[2] = ' ';
13.
14.      int Nap = (int) Volts;
15.      int Ostanek = (int)(Volts*1000);
16.      Ostanek = Ostanek - (Nap/10)*10000 - (Nap%10)*1000;
17.
18.      if (Nap>=10){
19.          WriteBuffer[3] = '0' + Nap/10;
20.          WriteBuffer[4] = '0' + Nap%10;
21.          WriteBuffer[5] = '.';
22.          WriteBuffer[6] = '0' + Ostanek/100;
23.          Ostanek = Ostanek - (Ostanek/100)*100;
24.          WriteBuffer[7] = '0' + Ostanek/10;
25.          WriteBuffer[8] = '0' + Ostanek%10;
26.          WriteBuffer[9] = '\r';
27.          WriteBuffer[10] = '\n';
28.          WriteBuffer[11] = '\0';
29.      }
30.      else {
31.          WriteBuffer[3] = '0' + Nap;
32.          WriteBuffer[4] = '.';
33.          WriteBuffer[5] = '0' + Ostanek/100;
34.          Ostanek = Ostanek - (Ostanek/100)*100;
35.          WriteBuffer[6] = '0' + Ostanek/10;
36.          WriteBuffer[7] = '0' + Ostanek%10;
37.          WriteBuffer[8] = '\r';
38.          WriteBuffer[9] = '\n';
39.          WriteBuffer[10] = '\0';
40.      }
41.      Write(WriteBuffer);
42.  }
```

Slika 4.23 : Definicija funkcije Set_OutputV

Funkcija **MaxV** (slika 4.24) je nadgrajena funkcija **Get_Range**. Slednja nam vrne vrednost območja, ki določa meje nastavljanja toka in napetosti na izhodih. Funkcija **MaxV** dobljeno

vrednost območja (0, 1 ali 2) poveže z maksimalno vrednostjo napetosti, ki jo je znotraj tega območja možno nastaviti. S tem smo dosegli, da klic funkcije **MaxV** v 5. vrstici kode na sliki 4.23 preveri, če je poslana vrednost napetosti znotraj meje območja.

```
01. int QL355TP::MaxV(int Output) {
02.     if(Output<1 || Output>3) {
03.         throw Ex_OpremaEA_Param();
04.     }
05.
06.     WriteBuffer = new char[WriteBufferSize];
07.
08.     WriteBuffer[0] = 'R';
09.     WriteBuffer[1] = 'A';
10.     WriteBuffer[2] = 'N';
11.     WriteBuffer[3] = 'G';
12.     WriteBuffer[4] = 'E';
13.     WriteBuffer[5] = '0' + Output;
14.     WriteBuffer[6] = '?';
15.     WriteBuffer[7] = '\r';
16.     WriteBuffer[8] = '\n';
17.     WriteBuffer[9] = '\0';
18.     Write(WriteBuffer);
19.     Sleep(450);
20.     size_t len = Read(ReadBuffer, ReadBufferSize);
21.     size_t len1 = len - 3;
22.     ReadBuffer[len] = '\0';
23.
24.     //if(len != 5) { throw Ex_OpremaEA_Logic(); }
25.
26.     switch(ReadBuffer[len1]) {
27.         case '0': {
28.             return 15;
29.                 }
30.         case '1': {
31.             return 35;
32.                 }
33.         case '2': {
34.             return 35;
35.                 }
36.         default: {
37.             throw Ex_OpremaEA_Logic();
38.                 }
39.                             } //switch(ReadBuffer[3])
40. }
```

Slika 4.24 : Definicija funkcije MaxV

5 Demonstracijski programi

V nadaljevanju smo na primerih demonstracijskih programov prikazali način uporabe razvitih knjižnic. Prva dva programa prikazujeta avtomatizirano izvajanje meritev na inštrumentu Hameg HM8118. Tretji program uporablja le funkcije iz knjižnice inštrumenta Keithley 2016p, medtem ko sta zadnja dva programa primer kombinirane uporabe funkcij dveh knjižnic: inštrumenta Keithley 2016p in napajalnika TTi QL355TP. Keithley 2016p je uporabljen kot generator vzbujevalnega sinusnega signala in kot inštrument za zajemanje izhodnega signala opazovanega električnega vezja. TTi QL355TP je napajalni vir vezja.

Nabor vseh parametrov in rezultatov programov shranjujemo v tekstovne datoteke, kar nam omogoča nadaljno obdelavo rezultatov meritev in njihov grafični prikaz.

5.1 Zajem vrednosti izbranega merilnega para pri spreminjanju frekvence

Namen prvega programa je merjenje parametrov pasivnih elementov električnega vezja pri preletu čez celotno območje frekvence vzbujanja (od 20 Hz do 200 kHz). Meritve smo izvedli z inštrumentom Hameg HM8118. Za vsako testno frekvenco izmerimo vrednosti nastavljenega merilnega para in ju shranimo v tekstovno datoteko. Uporabnik programa določi merilni par (kapacitivnost-upornost oziroma CR, konduktanca-susceptanca oziroma GB, induktivnost-upornost oziroma LR, impedanca-fazni_kot oziroma Ztheta, itn.).

```
01.  #include "Hameg.h"
02.  #include <windows.h>
03.  #include <iostream>
04.  #include <fstream>
05.
06.  void Hameg::SetGet_MeasurementValueAtChoosenFrequency(SetupCommands Measurement){
07.
08.      std::ofstream Text;
09.
10.      Text.open("text5.txt",ios::trunc);
11.
12.      Set_ManualRange(0);
13.      Set_PMOD(Measurement);
14.      OperationComplete();
15.
16.      Text << '\t' << '\t' << EnumToString(Get_PMOD())[0] << '\t';
17.      Text << EnumToString(Get_PMOD())[1] << '\n';
18.      Text << "Frequency" << '\n';
```

Slika 5.1 : Nastavljanje začetnih parametrov

Slika 5.1 prikazuje začetni del programa, kjer smo nastavili parametre inštrumenta. Najprej izklopimo ročno nastavljanje območja prikaza rezultata (funkcija **Set_ManualRange**) ter inštrument nastavimo na merjenje izbrane karakteristike. V tekstovno datoteko vpišemo oznake izbranega merilnega para.

V nadaljevanju smo za čim preciznejše nastavljanje frekvence celotno frekvenčno območje inštrumenta razdelili na pet manjših območij:

- od 20 Hz do 5 kHz s korakom 700 Hz,
- od 6 kHz do 9 kHz s korakom 800 Hz,
- od 10 kHz do 50 kHz s korakom 8000 Hz,
- od 50 kHz do 97 kHz s korakom 10000 Hz in
- od 100 kHz do 200 kHz s korakom 25000 Hz.

Vsako naslednjo vrednost frekvence (spremenjeno za določeni korak) pošljemo inštrumentu s funkcijo **Set_Frequency**. Funkciji **MainResult** in **SecondResult** nam vrneta vrednosti parametrov pri nastavljeni frekvenci. Dobljene vrednosti ter frekvenco pri kateri so vrednosti izmerjene, vpišemo v tekstovno datoteko.

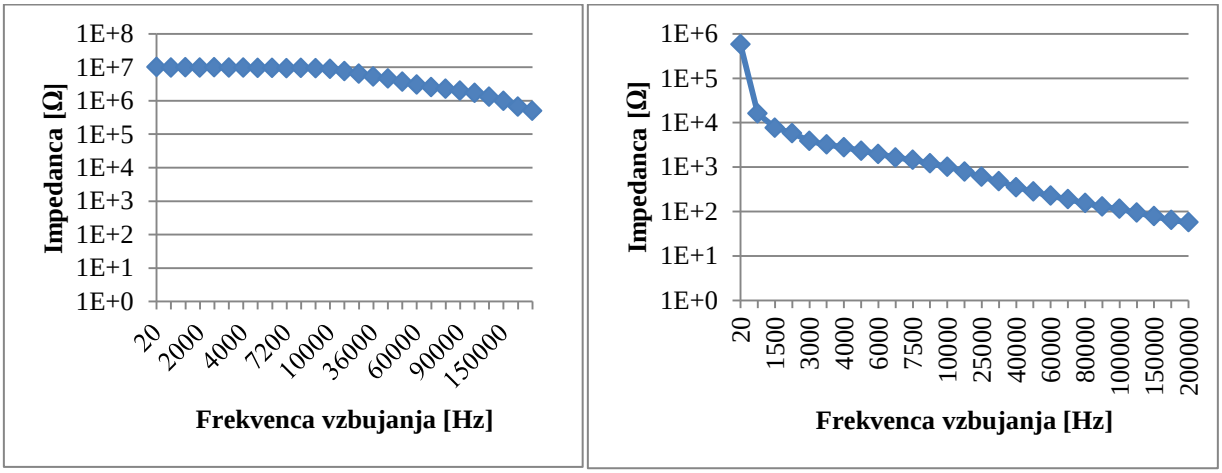
Slika 5.2 prikazuje programsko kodo za prvo frekvenčno območje.

```
01.  for (unsigned int j = 20; j < 5000; j = j + 700){
02.          Set_Frequency(j);
03.          //OperationComplete();
04.          Text << Get_Frequency();
05.          OperationComplete();
06.          Text << '\t' << '\t' << MainResult() << '\t';
07.          //OperationComplete();
08.          Text << SecondResult() << '\n';
09.          OperationComplete();
10.      }
```

Slika 5.2 : Prvo frekvenčno območje karakteristike

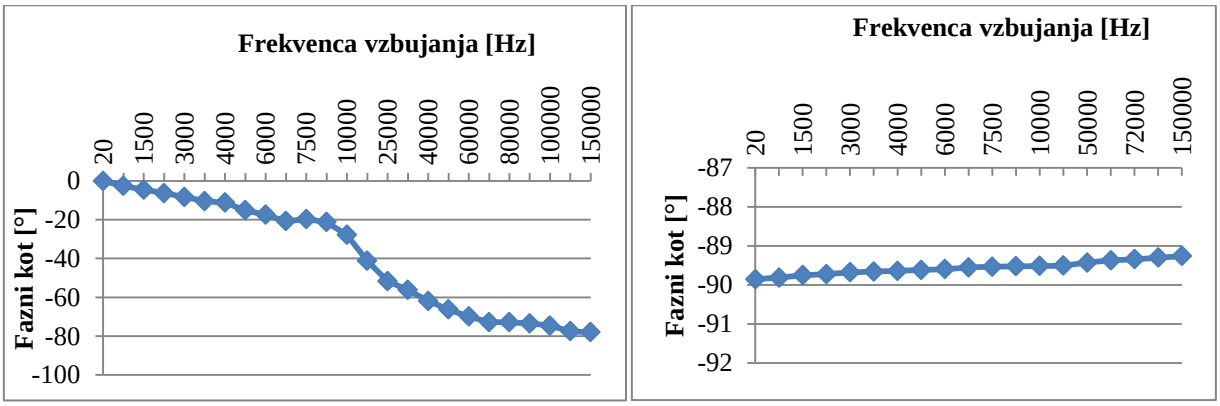
Zajem vrednosti izbranega merilnega para demonstrirajmo z vklopom pasivnega elementa (v našem primeru upora R = 1 MΩ in kondenzatora C = 15 nF) ter izbiro impedance in faznega kota kot želeni merilni par. Slika 5.3 prikazuje spremembo impedance upora in kondenzatorja pri večanju frekvence vzbujanja. Za boljšo preglednost grafov so vrednosti impedanc prikazane na logaritemski skali.

Pri uporu R = 1 MΩ vidimo, da se mu z večanjem frekvence do 10 kHz impedanca neznatno spreminja in ima vseskozi konstantno vrednost upornosti. Pri večjih frekvencah vrednost impedance in faznega kota začne upadat, kar je posledica vpliva parazitne kapacitivnosti. Na sliki 5.4(levo) opazimo povečano naraščanje faznega kota pri frekvencah, večjih od 10 kHz.



Slika 5.3 : Grafični prikaz frekvenčne odvisnosti impedance upora R = 1 MΩ (levo) in kondenzatorja C = 15 nF (desno) reproduciran iz shranjenih podatkov

Pri kondenzatorju C opazimo (slika 5.3 (desno)), da mu impedanca z večanjem frekvence na začetku hitro pade, takoj zatem se padanje umiri. Fazni kot nima konstantne vrednosti -90°, ampak se vrednost z naraščanjem frekvence manjša po absolutni vrednosti, kar je posledica parazitnih učinkov (slika 5.4 (desno)).



Slika 5.4 : Grafični prikaz frekvenčne odvisnosti faznega kota upora R = 1 MΩ (levo) in kondenzatorja C = 15 nF (desno) reproduciran iz shranjenih podatkov

5.2 Porazdelitev izmerjenih vrednosti pri večjem številu elementov

Namen programa je merjenje in zapisovanje vrednosti merilnega para pri različnih frekvencah ter vzporedno računanje nekaterih statističnih podatkov iz dobljenih vrednosti. Statistične metode [16] nam omogočajo določanje lastnosti zbranih podatkov ter povezavo in oblikovanje odnosov med dobljenimi vrednostmi. Bistvo našega programa je dobiti podatek o porazdelitvi izmerjenih vrednosti pri večjem številu merjencev.

Program je nadgradnja programa iz poglavja 5.1, kjer smo merili parametre pasivnega elementa pri različnih testnih frekvencah, kar je tudi osnova programiranja tega programa. Vendar je bilo potrebno poskrbeti, da trajanje izvajanja programa ni omejeno z številom merjencev ter, da pri vsakem merjencu izračunamo nove vrednosti statističnih podatkov.

Slika 5.5 prikazuje prvi del **while** zanke [9] kjer čakamo na zamenjavo merjenca. Uporabnik s pritiskom na katerokoli tipko (funkcija **_kbhit** [9] iz standardne C++ knjižnice *conio.h*) prekine izvajanje zanke. **While** zanka (10. vrstica) čaka na zamenjavo elementa, za katerega že imamo vse iskane vrednosti. **While** zanka (17. vrstica) se neprekinjeno izvaja vse dokler ne vstavimo naslednji element. S funkcijo **Sleep** (standardna C++ knjižnica *windows.h*) ustavimo izvajanje programa za izbrano število milisekund, da se po vklopu novega elementa stanje na inštrumentu umiri.

```
01. while(!found){
02.         if (Get_PMOD() != ZTheta) {
03.             Set_PMOD(ZTheta);
04.         }
05.         OperationComplete();
06.         Set_Frequency(20);
07.         OperationComplete();
08.         if (MainResult() < 800000000){
09.             first++;
10.             while (MainResult() < 800000000 && first != 1){
11.                 if (_kbhit()){
12.                     found = true;
13.                     break;
14.                 }
15.             }
16.             //Stop();Sleep(5000);
17.             while (MainResult() > 800000000 && !found){
18.                 if (_kbhit()){
19.                     found = true;
20.                     break;
21.                 }
22.             }
23.             Stop();Sleep(5000);
24.         }
```

Slika 5.5 : Zamenjava elementa

Statistične podatke [13, 14] računamo na podlagi izmerjenih vrednosti vseh merjencev za vse testne frekvence. Z uporabo vektorja vektorjev [9] (ang. Vector of vectors; standardna C++ knjižnica *vector*) smo dosegli, da se vrednosti vpisujejo v vektorje za vsako frekvenco posebej. To nam omogoča vpogled in dostop do vseh vrednosti potrebnih za izračun vsakega od izbranih statističnih podatkov.

Slika 5.6 prikazuje deklaracijo takšnih vektorjev ter primer definicije izračuna podatka standardnega odklona vzorca. Prvi korak izračuna standardnega odklona je izračun vzorčnega povprečja kar prikazujeta 12. in 14. vrstica programske kode. Funkcija **accumulate** (standardna C++ knjižnica *numeric*) sešteje vse vrednosti znotraj vektorja. V **for** zanki od vsakega števila odštejemo povprečje, to razliko kvadriramo in prištejemo vrednosti spremenljivke dev. V naslednji vrstici seštevek kvadratov oziroma vrednost spremenljivke dev delimo s številom vseh števil v vzorcu minus ena.

```
01. vector<vector<double> > VectorOfVectors;
02.
03. double MedianScore(vector<double> scores);
04. double LowerQuart(vector<double> scores);
05. double UpperQuart(vector<double> scores);
06. double StandardDev(vector<double> scores);
07.
08. double Hameg::StandardDev(vector<double> scores){
09.
10.     size_t size = scores.size();
11.
12.     double sum = std::accumulate( scores.begin(), scores.end(), 0.0 );
13.
14.     sum = sum / size;
15.
16.     double dev = 0;
17.
18.     for (size_t i = 0; i < size; ++i){
19.         dev = dev + pow (scores[i] - sum, 2);
20.     }
21.
22.     dev = sqrt (dev / (size-1));
23.
24.     if (size == 1){
25.         dev = 0;
26.     }
27.
28.     return dev;
29. }
```

Slika 5.6 : Deklaracija vektorjev in definicija standardnega odklona vzorca

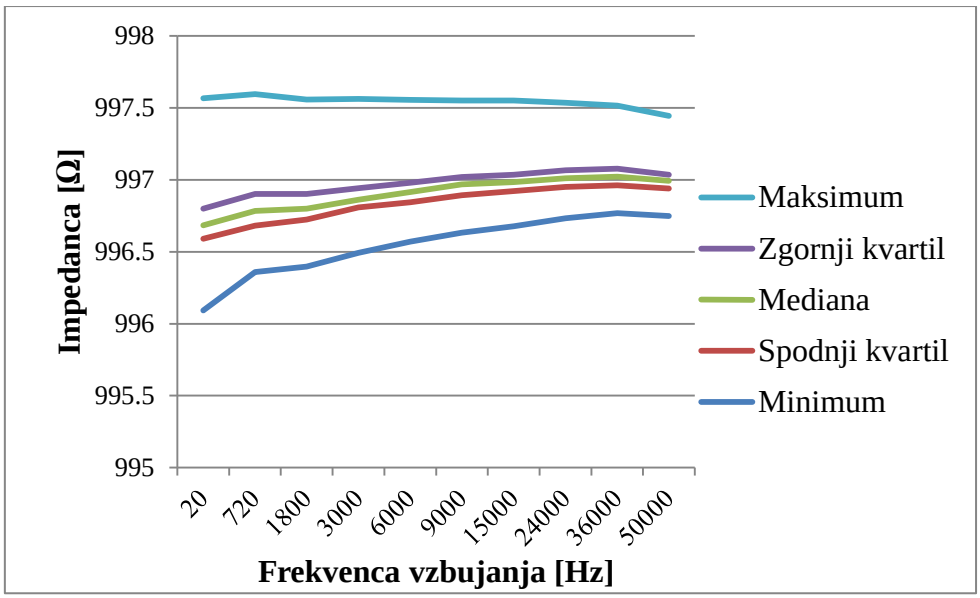
Vpisovanje vrednosti v vektorje prikazuje slika 5.7. Pri izračunu vrednosti za prvi vklopljeni element (spremenljivka i = 1) definiramo vektor vektorjev za vsako frekvenco (19. vrstica).

Vrednosti ostalih elementov se zapovrstjo vpisujejo v definiran vektor. Ta vektor uporabimo kot parameter funkcij za izračun statističnih podatkov.

```
01.  void Hameg::SetGet_TriggeredMeasurementsOfElements(SetupCommands Measurement){
02.
03.          vector<double> row;
04.
05.          if (!found && MainResult() > 0 && MainResult() < 800000000){
06.              OperationComplete();
07.              Set_PMOD(Measurement);
08.              OperationComplete();
09.              i++;
10.              k = 0;
11.              Text << i;
12.              cout << i;
13.              for (double j = 20; j <= 200000; j = 2*j + 700){
14.                  Set_Frequency(j);
15.                  OperationComplete();
16.                  Text << '\t' << Get_Frequency();
17.                  OperationComplete();
18.
19.                  if (i == 1){
20.                  VectorOfVectors.push_back(row);
21.                  }
22.
23.                  VectorOfVectors[k].push_back(MainResult());
24.
25.                  sum = std::accumulate( VectorOfVectors[k].begin(), VectorOfVectors[k].end(), 0.0 );
26.                  Text << '\t' << '\t' << MainResult();
27.                  Text << '\t' << '\t' << sum/i;
28.                  Text << '\t' << '\t' << MedianScore(VectorOfVectors[k]);
29.                  Text << '\t' << '\t' << LowerQuart(VectorOfVectors[k]);
30.                  Text << '\t' << '\t' << UpperQuart(VectorOfVectors[k]);
31.                  Text << '\t' << '\t' << StandardDev(VectorOfVectors[k]) << '\n';
32.
33.                  k++;
34.              }
35.          }
36.  }
```

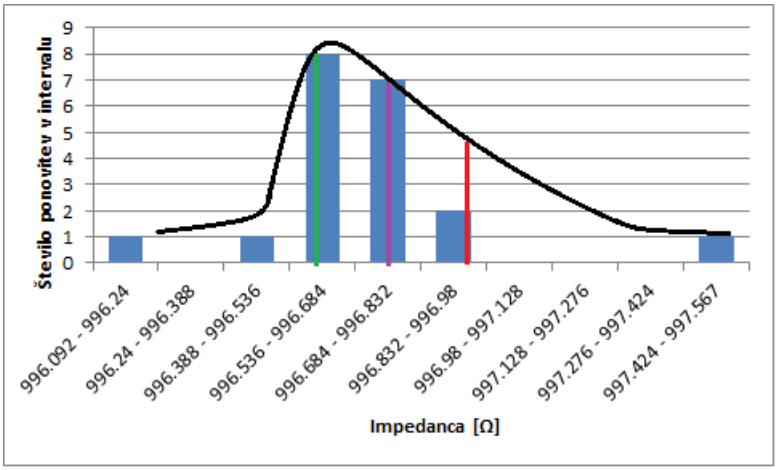
Slika 5.7 : Vpisovanje vrednosti v vektor vektorjev in v tekstovno datoteko

Za najboljši grafični prikaz dobljenih podatkov smo se ponovno odločili za izbiro merilnega para impedance in faznega kota. Na sliki 5.8 je prikazan graf porazdelitve izmerjenih vrednosti impedance uporov. Graf je sestavljen iz izmerjenih podatkov za 20 uporov z upornostjo 1000 Ω. Termin “povzetek petih števil” (ang. Five-number summary) se nanaša na vrednosti spodnjega in zgornjega kvartila, mediane ter maksimuma in minimuma. Primerjavo teh vrednosti z vrednostmi vseh izmerjenih impedanc uporov uporabljamo za proučevanje oblike porazdelitve dobljenih vrednosti.



Slika 5.8 : Porazdelitev izmerjenih vrednosti impedance upora R = 1 kΩ oziroma "Povzetek petih števil"

Z uporabo funkcije Frequency v programu Microsoft Excel 2010, smo vrednosti impedanc vseh 20 merjencev izmerjenih pri frekvenci 20 Hz, razdelili med osem enakih intervalov in sestavili histogram [15] (slika 5.9). Na graf smo vrisali še linije, ki prikazujejo srednjo vrednost (vijolično), mediano (zeleno) in modus (rdeče).



Slika 5.9 : Histogram za izmerjene vrednosti impedance uporov R = 1 kΩ pri frekvenci 20 Hz

5.3 Zajem amplitudnega spektra s parametrizacijo frekvence vzbujanja

Namen programa je zajemanje amplitudnega spektra pri spreminjanju frekvence vzbujanja izbranega vezja. Uporabniku je tokrat naložena izbira štirih parametrov za izvajanje meritev. Frekvenčni korak meritve in amplituda določata vzbujevalni signal. Te vrednosti označujejo tudi minimalno ali začetno vrednost frekvence in amplitude signala. Za vsako kombinacijo teh vrednosti se zajame ena meritev amplitudnega spektra.

Inštrument Keithley 2016p omogoča merjenje amplitude pri do 1023 frekvenčnih odsekih naenkrat. Te odseke imenujemo koši (ang. bins). Druga dva parametra, ki ju uporabnik določi sta velikost in število košev. Produkt množenja frekvenčnega koraka, izbrane velikosti in števila košev je maksimalna frekvenca, ki jo bomo dosegli. Ta vrednost ne sme presegati vrednosti maksimalne frekvence inštrumenta. Na ta način določamo območje in ločljivost frekvenc, ki jih merimo in prikazujemo na grafu amplitudnega spektra.

Preden se posvetimo postopku sestavljanja programske kode programa, povejmo nekaj o funkcijah **Set_ListOfFrequenciesAnalized, Get_AmplitudeOfListFrequencies** in **Set_Freqlist**. Te funkcije so podprogrami ustvarjeni za delovanje našega programa. Funkcija **Set_Freqlist** uporablja tri podane parametre programa za delovanje − začetno frekvenco, velikost ter število košev. Naloge funkcije so:

- iz podanih vrednosti parametrov določiti območje in ločljivost frekvenc, ki jih bomo merili,
- preveriti, če je maksimalna dosežena frekvenca manjša od maksimalne frekvence inštrumenta ter
- ustvariti 50 členske (omejitev inštrumenta) vektorje v katerih so zapisane vrednosti merjenih frekvenc.

Funkcija **Set_ListOfFrequenciesAnalized** nam omogoča, da te 50 členske vektorje enega za drugim pošiljamo inštrumentu. Ustvarimo še vektor, ki bo vseboval vrednosti vseh frekvenc ter vejice, ki te vrednosti ločujejo. Listo vrednosti amplitud za vse poslane vrednosti frekvence dobimo v obliki vektorja z uporabo funkcije **Get_AmplitudeOfListFrequencies**.

Definicija funkcije je prikazana na sliki 5.10. Delovanje in namen večine uporabljenih funkcij smo opisali v poglavju 4.2.2. V nadaljevanju smo razložili vrstni red njihovega izvajanja, izbrane vrednosti parametrov in postopek programiranja omenjenih podprogramov.

Najprej smo z uporabo funkcij **ResetInstrument** in **ClearStatus** ponastavili inštrument na standardne (ang. default) nastavitve in vrednosti registrov na 0. Izklopili smo neprekinjeno sprožanje meritev s funkcijo **Set_ContinuousInitiationState**. Število operacij, ki naj jih inštrument izvaja pri naslednji sproženi meritvi smo nastavili na vrednost 1 s funkcijo **Set_OperationLoopsNumInTrigger**. S tem smo dosegli, da inštrument vsakič, ko preide v OQAS ob sproženi meritvi s funkcijo **OneTriggerCycle** osveži nastavitve.

```
01. void Keithley::SetGet_FrequencyListProgrammingExampleBins(unsigned int Freq,double Amplitude,
02.                                                           unsigned int Bin,unsigned int NumOfBins){
03.     ResetInstrument();
04.     ClearStatus();
05.     Set_ContinuousInitiationState(0);
06.     Set_OperationLoopsNumInTrigger(1);
07.     Set_MeasurementFunction(Distortion);
08.     Set_DistInputSignalFreq(20);
09.     //Set_SineWave50OhmOutput(Freq,Amplitude);
10.     //Set_SineWave600OhmOutput(Freq,Amplitude);
11.     Set_SineWaveHighImpedanceOutput(Freq,Amplitude);
12.     //Set_PulseWaveformShapeChannel2();
13.     //Set_InvertedSineWaveformShapeChannel2();
14.     Set_OutputState(1);
15.     OneTriggerCycle();
16.     Get_OperationComplete();
17.
18.     Times = NumOfBins/50;
19.     if (NumOfBins%50 != 0) Times = Times + 1;
20.
21.     std::fstream Text;
22.     Text.open("text52.txt",ios::app);
23.     Text << '\t' << "Frequency" << '\t' << "Amplitude" << '\t' << "Freq_Step" << '\n';
24.     Text << '\t' << Get_SineWaveOutputFreq() << '\t' << Get_SineWaveOutputAmpl() << '\t' << Bin << '\n';
25.     Text << '\t' << "FrequencyList" << '\t' << "AmplitudeList" << '\n';
26.
27.     for(Repeat = 0; Repeat < Times; Repeat++){
28.         Set_Freqlist(Freq,Bin,NumOfBins);
29.         Set_ListOfFrequenciesAnalized(Freqlist);
30.         Get_AmplitudeOfListFrequencies();
31.     }
32.     double Peak;    Peak = FULL_AmplitudeOfListFrequencies[0];
33.     for (unsigned int i = 0; i < NumOfBins-1; i++){
34.         if (FULL_AmplitudeOfListFrequencies[i] > FULL_AmplitudeOfListFrequencies[i+1] && i != 0 &&
35.             FULL_AmplitudeOfListFrequencies[i] > Peak){
36.             Peak = FULL_AmplitudeOfListFrequencies[i];
37.         }
38.     }
39.     for (unsigned int i = 0; i < NumOfBins; i++){
40.         Text <<'\t'<< FULL_ListOfFrequenciesAnalized[i] <<'\t'<< FULL_AmplitudeOfListFrequencies[i] - Peak <<'\n';
41.     }
42.
43.     Text.close();
44. }
```

Slika 5.10 : Definicija funkcije zajema amplitudnega spektra pri spreminjanju frekvence vzbujanja vezja

Kot parameter funkcije **Set_MeasurementFunction** smo izbrali merilno funkcijo, ki meri popačenje (ang. distortion). Tip popačenja pri tovarniških nastavitvah je THD kar pomeni, da ponovni klic funkcije nastavljanja tipa popačenja ni potreben.

Preden vklopimo generator izhodnega signala je potrebno določiti njegove parametre. Vrednost izhodne impedance določimo z izbiro ene od treh funkcij **Set_SineWaveOutput**. Frekvenca in amplituda, ki sta parametra teh funkcij sta tudi parametra programa. **N**june vrednosti določimo vsakič pred začetkom izvajanja meritev. Ko smo nastavili parametre izhodnega signala vklopimo njegov generator. Z uporabo funkcije **OneTriggerCycle** osvežimo nastavitve.

V tekstovno datoteko vpišemo izbrane vrednosti frekvence in amplitude izhodnega signala ter frekvenčni korak pri izvajanju meritve. V **for** zanki (27. vrstica) vpišemo vrednosti zajetih frekvenc v 50 členske vektorje s funkcijo **Set_Freqlist**. Te vektorje pošljemo inštrumentu s funkcijo **Set_ListOfFrequenciesAnalized** ter z uporabo funkcije **Get_AmplitudeOfListFrequencies** preberemo listo dobljenih amplitud. Njihovo izvajanje ponavljamo dokler ne dobimo vrednosti amplitud za vse frekvence. V tekstovno datoteko vpišemo vrednosti vseh frekvenc in amplitud zajetih v meritvi.

Rezultat programske kode omenjenih podprogramov sta vektorja oziroma listi amplitud in frekvenc pri katerih meritve izvajamo. Razlago postopka njihovega programiranja smo začeli pri funkciji **Set_Freqlist**. Njena definicija je prikazana na sliki 5.11.

```
01. void Keithley::Set_Freqlist(unsigned int Freq, unsigned int Bin, unsigned int NumOfBins){
02.     if (Bin*NumOfBins > 20000-Freq){throw Ex_OpremaEA_Param();}
03.     unsigned int Resize = 50;
04.     if (Repeat == Times-1 && NumOfBins%50 != 0){Resize = NumOfBins%50;}
05.     Points.resize (Resize);
06.     Channels = 0;
07.     Size = 0;
08.     for(unsigned int i = Repeat*50*Bin; i < (Repeat+1)*50*Bin && NumOfBins > (i/Bin) ; i = i + Bin){
09.         Channels = i + 100;
10.         Points[Size] = Channels;
11.         Size++;
12.     }
13.     unsigned int Value = 0;
14.     WriteBuffer = new char[WriteBufferSize];
15.     WriteBuffer[0] = '0';
16.     Size = 0;
17.     Channels = 0;
18.     int Comma = 0;
19.     for(unsigned int j = 0; j < Resize; j++){
20.         Value = Points[j];
21.         if (Value != 0 && Value < 20000){
22.             for (int i = 5; i > 0; i--){
23.                 WriteBuffer[Size+Comma+i] = '0' + Value%10;
24.                 Channels = Channels++;
25.                 Value = Value/10;
26.             }
27.             if (j != 50-1){
28.                 WriteBuffer[Channels+Comma+1] = ',';
29.             }
30.             Size = Channels;
31.             Comma++;
32.         }
33.         else {
34.             j = 50;
35.         }
36.     }
37.     WriteBuffer[Comma+Channels] = '\0';
38.     Freqlist = WriteBuffer;
39. }
```

Slika 5.11 : Definicija funkcije Set_Freqlist

V prvi **if** zanki smo določili pogoj, da pomnožena vrednost podanih parametrov, velikosti frekvenčnega koraka in števila korakov, ni večja od maksimalne frekvence. Namen druge **if** zanke je določitev števila členov v vektorju, ki je odvisen od podanega števila frekvenčnih korakov. V nadaljevanju vektorju Points spremenimo velikost (ang. Resize) na dobljeno število členov ter določimo njihove vrednosti, ki se medseboj razlikujejo za velikost frekvenčnega koraka. V **for** zanki (19. vrstica) vrednosti frekvenc znotraj vektorja skupaj z vejicami vpišemo v medpomnilnik pisanja WriteBuffer. Na konec dodamo oznako za zaključek znakovnega niza, katerega shranimo v spremenljivko Freqlist. Spremenljivka Freqlist je podatkovnega tipa string. Ta znakovni niz oziroma spremnljivka Freqlist je tudi parameter funkcije **Set_ListOfFrequenciesAnalized** (Slika 5.12).

```
01. void Keithley::Set_ListOfFrequenciesAnalized(std::string Freqlist){
02.     ListNumElem = 0;
03.     WriteBuffer = new char[WriteBufferSize];
04.     for (int i = 0; Freqlist[i] != '\0'; i++){
05.         if (Freqlist[i] == ','){ListNumElem++;}
06.     }
07.     if (ListNumElem > 49){throw Ex_OpremaEA_Param();}
08.     ListNumElem = ListNumElem+1;
09.     ListOfFrequenciesAnalized.resize (ListNumElem);
10.     Size = 0;
11.     size_t j = 0;
12.     size_t i = 0;
13.     for (i = 0; Freqlist[i] != '\0'; i++){
14.         char c = Freqlist[i];
15.         WriteBuffer[i-j] = c;
16.         if (c == ','){
17.             unsigned int Bins = atoi (WriteBuffer);
18.             ListOfFrequenciesAnalized[Size] = Bins;
19.             Size = Size++; j = i+1;
20.         }
21.     }
22.     unsigned int Bins = atoi (WriteBuffer);
23.     ListOfFrequenciesAnalized[Size] = Bins;
24.     FULL_ListOfFrequenciesAnalized.resize (50*Times);
25.     for (unsigned int i = 0; i < Size+1; i++){
26.         FULL_ListOfFrequenciesAnalized[i+50*Repeat] = ListOfFrequenciesAnalized[i];
27.     }
28.     Size = 0;
29.     delete[](WriteBuffer);
30.     WriteBuffer = new char[WriteBufferSize];
31.     WriteBuffer[0] = ':';WriteBuffer[1] = 'D';WriteBuffer[2] = 'I';WriteBuffer[3] = 'S';WriteBuffer[4] = 'T';
32.     WriteBuffer[5] = ':';WriteBuffer[6] = 'P';WriteBuffer[7] = 'E';WriteBuffer[8] = 'A';WriteBuffer[9] = 'K';
33.     WriteBuffer[10] = ':';WriteBuffer[11] = 'L';WriteBuffer[12] = 'I';WriteBuffer[13] = 'S';WriteBuffer[14] = 'T';
34.     WriteBuffer[15] = ' ';
35.     Size = 16;
36.     Channels = 0;
37.     for (int j = 0; Freqlist[j] != '\0'; j++){
38.         WriteBuffer[Size+j] = Freqlist[j];
39.         Channels++;
40.     }
41.     WriteBuffer[Size+Channels] = '\r';
42.     WriteBuffer[Size+Channels+1] = '\0';
43.     Write(WriteBuffer);
44. }
```

Slika 5.12 : Definicija funkcije Set_ ListOfFrequenciesAnalized

Pri pošiljanju liste frekvenc inštrument zahteva maximalno 50 naenkrat poslanih vrednosti in vejice, ki te vrednosti ločijo (spremenljivka Freqlist**)**. Težava se pokaže, ko želimo dostopati do posameznih členov znotraj spremenljivke Freqlist. Podatkovni tip string je znakovni niz, ki ne loči razlike med številko, črko ali vejico. Nemogoče je vedeti kje se vrednost določene frekvence konča in druge začne. Lista dobljenih vrednosti amplitud je smislena edino skupaj z popisom vseh poslanih frekvenc ter zaradi tega potrebujemo drugačno obliko spremnljivke Freqlist.

Glavni produkt funkcije **Set_ListOfFrequenciesAnalized** je vektor FULL_ListOfFrequenciesAnalized, ki je sestavljen iz vseh poslanih vektorjev oziroma liste vseh vrednosti poslanih frekvenc. V obliki vektorja je lista frekvenc primerna za zapis v tekstovno datoteko, ker ima vsak člen vektorja zraven vrednosti tudi indeksno vrednost. Na ta

način je s poljubno spremenljivko možno pristopiti do vsakega člena vektorja (FULL_ListOfFrequenciesAnalized[i]).

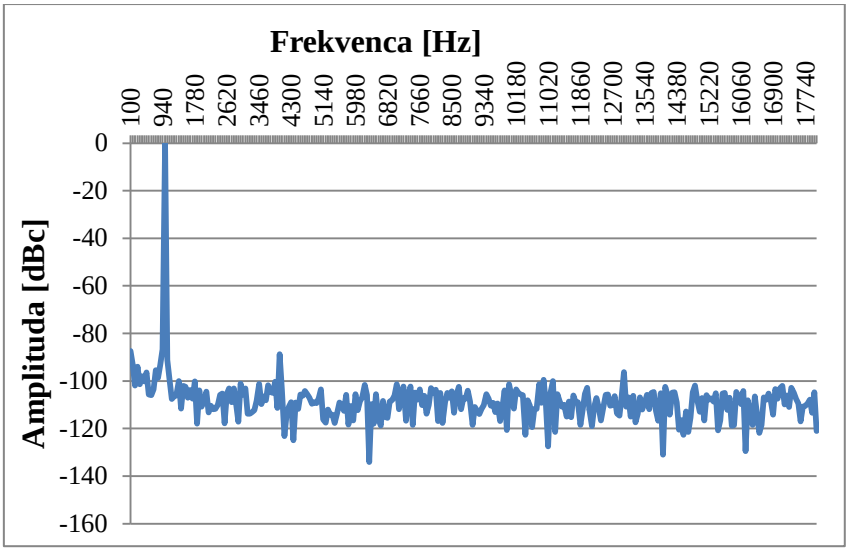
S funkcijo **Get_AmplitudeOfListFrequencies** (Slika 5.13) preberemo vrednosti dobljenih amplitud ter jih zapišemo v vektor FULL_AmplitudeOfListFrequencies. Rezultate končane meritve oziroma vse vrednosti znotraj vektorjev FULL_ListOfFrequenciesAnalized in FULL_AmplitudeOfListFrequencies, vzporedno (s spreminjanjem vrednosti spremenljivke i) prepišemo v tekstovno datoteko (Slika 5.10; 39. vrstica).

Število ponovitev izvajanja omenjenih kombiniranih funkcij v meritvi, je enako celoštevilskemu kvocientu deljenja števila frekvenčnih korakov s številom 50. V primeru, da je ostanek deljenja različen od 0 se število ponovitev poveča za ena.

```
01. void Keithley::Get_AmplitudeOfListFrequencies() {
02.     Write(":DIST:PEAK:LIST:DATA?\r");
03.     Sleep(1300);
04.     size_t len1 = Read(ReadBuffer, ReadBufferSize);
05.     ReadBuffer[len1] = '\0';
06.
07.     Size = 0;
08.     size_t j = 0;
09.
10.     WriteBuffer = new char[WriteBufferSize];
11.     AmplitudeOfListFrequencies.resize (ListNumElem);
12.
13.     for(size_t i = 0; i < len1; i++){
14.         char c = ReadBuffer[i];
15.         WriteBuffer[i-j] = c;
16.         if (c == ',' || c == '\r'){
17.             double Bins = atof (WriteBuffer);
18.             Set_OPC();
19.             AmplitudeOfListFrequencies[Size] = Bins;
20.             Size = Size++;
21.             j = i+1;
22.         }
23.     }
24.
25.     FULL_AmplitudeOfListFrequencies.resize (50*Times);
26.
27.     for (unsigned int i = 0; i < Size; i++){
28.         FULL_AmplitudeOfListFrequencies[i+50*Repeat] = AmplitudeOfListFrequencies[i];
29.     }
30. }
```

Slika 5.13 : Definicija funkcije Get_AmplitudeOfListFrequencies

Zajem amplitudnega spektra s parametrizacijo frekvence vzbujevalnega signala smo prikazali na primeru direktnega merjenja vzbujevalnega signala (brez vmesnega električnega vezja). Vzbujevalni signal je sinusne oblike, frekvenca vzbujanja je 1 kHz in amplituda vzbujanja 0,5 V.



Slika 5.14 : Zajem amplitudnega spektra s parametrizacijo frekvence vzbujanja

Graf na sliki 5.14 prikazuje rezultat meritve. Ordinatna os predstavlja amplitudo v normiranih decibelih na nosilni signal (dBc), abcisna os predstavlja frekvenco meritve [Hz]. Začetna frekvenca oziroma frekvenčni korak merjenja amplitudnega spektra je enak 60 Hz in število košev je 300. Na sliki opazimo, da amplitudni spekter zavzame maksimalno vrednost, ko je frekvenca meritve enaka frekvenci sinusa vzbujevalnega signala. Pri vseh ostalih frekvencah je vrednost spektra 100 dB nižja od nosilnega signala.

5.4 Zajem amplitudnega spektra s parametrizacijo amplitude vzbujanja

Produkt programa je amplitudni spekter pri parametrizaciji amplitude vzbujanja. Parametri potrebni za izvajanje meritev so velikost koraka spremembe napajalne napetosti vezja, frekvenca in amplituda signala ter število in velikost košev.

Definicija funkcije je prikazana na sliki 5.15. Sestavljena je iz nastavitvenih in nadzornih funkcij napajalnika TTi QL355TP in inštrumenta Keithley. Delovanje in namen večine uporabljenih funkcij smo opisali v poglavju 4.2.2 in 4.2.3. V nadaljevanju poglavju smo razložili vrstni red njihovega izvajanja in izbrane vrednosti parametrov.

V tretji in četrti vrstici smo uporabili kazalca x in k, ki se navezujeta na strukture naših dveh inštrumentov. S tem smo določili, da je uporaba katerekoli funkcije inštrumentov možna le ob uporabi kazalca, ki kaže na strukturo tega inštrumenta. Za primer opišimo uporabo funkcije **ResetInstrument.** Klic te funkcije je slučajno enak za oba inštrumenta, torej program pri klicu funkcije brez uporabe kazalcev ne bi vedel kateri inštrument je potrebno ponastavit. V primeru, ko imamo več popolnoma enakih inštrumentov je uporaba kazalcev nujna.

```
01. #include <PowerSupply.h>
02. #include <Keithley.h>
03. QL355TP x(6);
04. Keithley k(3);
05.
06. void Primeri::SetGet_FrequencyListVoltageInput(double VoltDifference,unsigned int Freq,
07. double AmplitudeDifference,unsigned int Bin,unsigned int NumOfBins){
08.     x.ResetInstrument();
09.     x.ClearStatus();
10.     x.OperationComplete();
11.     x.Set_OutputV(1,0);
12.     x.Set_OutputV(2,0);
13.     x.Set_StepSizeVoltage(1,VoltDifference);
14.     x.Set_StepSizeVoltage(2,VoltDifference);
15.
16.     while (x.Get_OutputV(1) < 15.01){
17.         unsigned int Ampl = 0;
18.         Sleep(2000);
19.         //x.Set_AllOnOff(1);
20.         x.Set_OnOff(1,1);
21.         x.Set_OnOff(2,1);
22.         Sleep(5000);
23.         k.Set_SineWaveOutputAmplHighZ(0.0);
24.         k.Set_SineWaveOutputAmpl(0.0);
25.         while (k.Get_SineWaveOutputAmpl() < (4 - AmplitudeDifference)){
26.             Ampl++;
27.             k.SetGet_FrequencyListProgrammingExampleBins(Freq,Ampl*AmplitudeDifference,Bin,NumOfBins);
28.             k.Get_OperationComplete();
29.         }
30.         k.Text.open("text52.txt",ios::app);
31.         k.Text << "Voltage" << '\t' << x.Get_OutputV(1) << '\t';
32.         x.OperationComplete();
33.         k.Text << -x.Get_OutputV(2) << '\n' << '\n';
34.         x.OperationComplete();
35.         //x.Set_AllOnOff(0);
36.         x.Set_OnOff(1,0);
37.         x.Set_OnOff(2,0);
38.         Sleep(1000);
39.         k.Text.close();
40.         x.IncVStepSize(1);
41.         x.IncVerifyStepSizeV(2);
42.         Sleep(1000);
43.     }
44. }
```

Slika 5.15 : Definicija funkcije zajema ampltudnega spektra s parametrizacijo amplitude vzbujanja

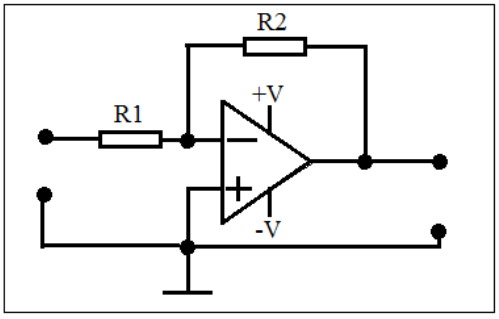
Na začetku funkcije **SetGet_FrequencyListVoltageInput** ponastavimo napajalnik na tovarniške nastavitve ter zbrišemo vrednosti registrov s funkcijami **ResetInstrument** in **ClearStatus**. Nastavimo začetne vrednosti napetosti na oba izhoda s funkcijo **Set_OutputV**. S funkcijo **Set_StepSizeVoltage** določimo vrednost koraka parametrizacije. **While** zanka [10] v

16. vrstici se neprekinjeno ponavlja dokler vrednost napetosti na prvem izhodu ne doseže maksimalno vrednost. Maksimalno vrednost smo tukaj omejili na 15 V. Večja napetost bi uničila operacijski ojačevalnik, katerega smo uporabili za izvajanje naših meritev. S funkcijo **Set_OnOff** aktiviramo izhoda. Začetno vrednost amplitude signala nastavimo s funkcijo **Set_SineWaveOutputAmpl**. Izvajanje **while** zanke (v vrstici 25) se konča, ko vrednost amplitude signala doseže maksimalno vrednost. Ta vrednost je pri inštrumentu Keithley omejena na 2 V_{rms} pri nizki izhodni impedanci generatorja signala ter 4 V_{rms} pri visoki impedanci.

Klic funkcije **SetGet_FrequencyListProgrammingExampleBins** vrne amplitudni spekter pri spreminjanju amplitude vhodnega signala. Ko dobimo spektre za vse amplitude, rezultate skupaj s trenutno vrednostjo napajalne napetosti vpišemo v tekstovno datoteko. Izhoda deaktiviramo in zvišamo vrednost napetosti za korak spreminjanja. Pri izvajanju meritev je potrebno skrbeti za pravilni vrstni red izvajanja funkcij vseh uporabljenih inštrumentov.

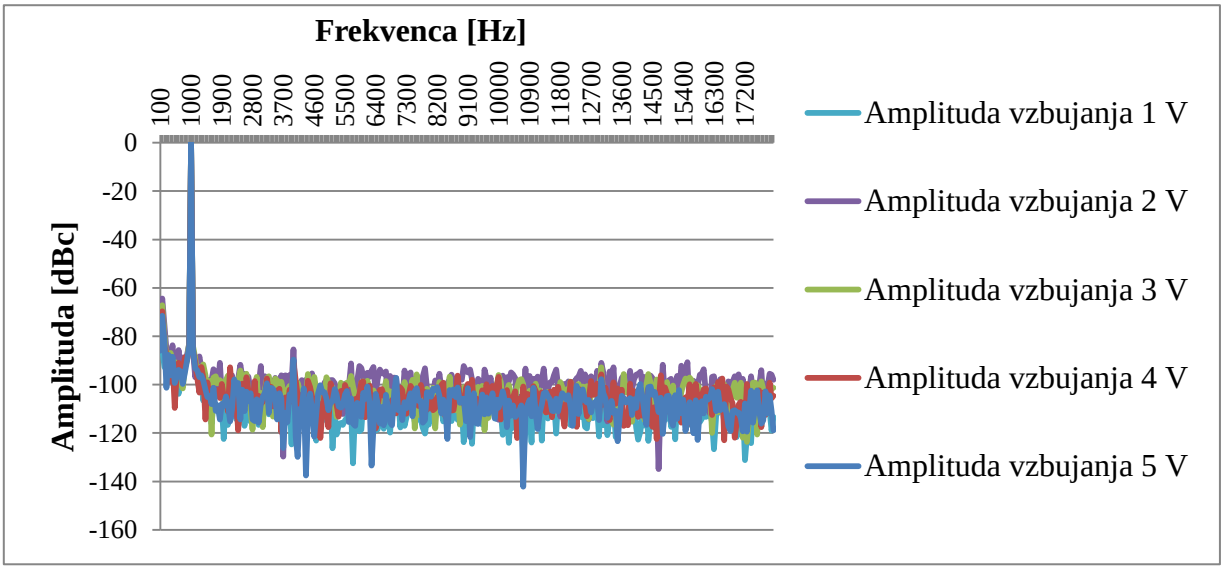
Uporaba funkcije **Sleep** ustavi izvajanje programa za izbrano število milisekund. S tem zagotovimo čas za osvežitev novih nastavitev na inštrumentih. Ko dosežemo maksimalno vrednost napetosti na prvem izhodu se prekine izvajanje **while** zanke in je meritev končana. Vse izmerjene vrednosti so shranjene v tekstovno datoteko za nadaljnjo obdelavo.

Za demonstracijo zajema amplitudnega spektra s parametrizacijo amplitude vzbujanja smo izbrali zelo preprosto vezje. Gre za meritev vzbujevalnega signala po prehodu skozi vezje invertirajočega ojačevalnika (slika 5.16).



Slika 5.16 : Invertirajoči ojačevalnik

Na levi sponki priklopimo vzbujevalni signal, na desnih sponkah pa zajamemo izhodni signal. Zgornja in spodnja sponka predstavljata napajalni sponki operacijskega ojačevalnika [16] (npr. +12 V in –12 V). Ojačanje vhodnega signala je enako $A_u = -\frac{R1}{R2}$. Uporabili smo enaka upora (R1 = R2) in s tem dobili ojačanje A_u = -1. Vrednost amplitude vzbujanja je v enoti $V_{rms} = \frac{Vp}{\sqrt{2}}$. Na izhodu dobimo invertiran vhodni signal z ojačanjem 1.



Slika 5.17 : Zajem amplitudnega spektra s parametrizacijo amplitude vzbujanja

Slika 5.17 prikazuje spekter pri različnih amplitudah vzbujevalnega signala. Frekvenca vzbujanja je 1 kHz, frekvenčni korak 60 Hz in število košev 300. Napajalna napetost je ±12 V. Dokler je amplituda vzbujevalnega signala krepko znotraj razpona napajalne napetosti ne pride do znatnega popačenja signala in je spekter približno enak za vse amplitude.

5.5 Zajem karakteristike popačenja v odvisnosti od napajalne napetosti

Produkt programa je karakteristika popačenja vhodnega signala v odvisnosti od napajalne napetosti. Definicija funkcije oziroma programa je prikazana na sliki 5.18. V programski kodi funkcije opazimo, da razen uporabe spremenljivk seek in found, ni drugih novitet v primerjavi s kodo iz programa v poglavju 5.4.

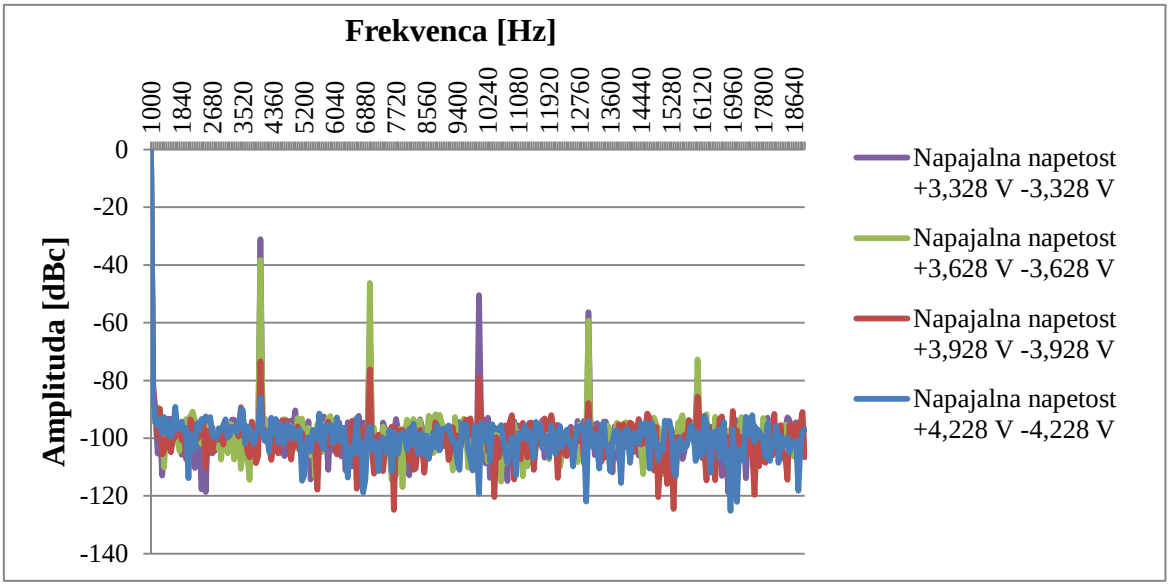
```
01. void Primeri::SetGet_VoltageForAmplitude(double VoltDifference,unsigned int Freq,
02.                 double Amplitude,unsigned int Bin,unsigned int NumOfBins){
03.     k.Set_OutputState(0);
04.     x.ResetInstrument();
05.     x.ClearStatus();
06.     x.OperationComplete();
07.     x.Set_OutputV(1,Amplitude*sqrt(2.0)+0.5);
08.     x.Set_OutputV(2,Amplitude*sqrt(2.0)+0.5);
09.     x.OperationComplete();
10.     x.Set_StepSizeVoltage(1,VoltDifference);
11.     x.Set_StepSizeVoltage(2,VoltDifference);
12.     x.OperationComplete();
13.
14.     bool found = false;
15.     while (x.Get_OutputV(1) < (Amplitude*sqrt(2.0) + 3) && !found){
16.         unsigned int Ampl = 0;
17.         Sleep(2000);
18.         //x.Set_AllOnOff(1);
19.         x.Set_OnOff(1,1);
20.         x.Set_OnOff(2,1);
21.         Sleep(5000);
22.         k.SetGet_FrequencyListProgrammingExampleBins(Freq,Amplitude,Bin,NumOfBins);
23.         k.Get_OperationComplete();
24.         if (k.seek == 0) found = true;
25.         k.Set_OutputState(0);
26.         k.Text.open("text70.txt",ios::app);
27.         k.Text << "Voltage" << '\t' << x.Get_OutputV(1) << '\t';
28.         x.OperationComplete();
29.         k.Text << -x.Get_OutputV(2) << '\n' << '\n';
30.         x.OperationComplete();
31.         //x.Set_AllOnOff(0);
32.         x.Set_OnOff(1,0);
33.         x.Set_OnOff(2,0);
34.         Sleep(1000);
35.         k.Text.close();
36.         x.IncVStepSize(1);
37.         x.OperationComplete();
38.         x.IncVStepSize(2);
39.         x.OperationComplete();
40.         Sleep(3000);
41.     }
42. }
```

Slika 5.18 : Definicija funkcije zajema karakteristike popačenja v odvisnosti od napajalne napetosti

Popačenje je v večini primerov neželeni pojav. V našem primeru je posledica omejitve napajalne napetosti vezja. Ko opazujemo izhodni signal vezja pri spreminjanju napajalne napetosti so meritve popačenja zelo uporabne. Napajalna napetost je neposredno povezana z maksimalno (in minimalno) napetostjo, ki jo lahko neko vezje generira. Pri operacijskem ojačevalniku je napajalna napetost tipično za nekaj sto milivoltov večja od največje (manjša od najmanjše) napetosti, ki jo lahko le-ta generira.

Za prikaz zajema karakteristike popačenja v odvisnosti od napajalne napetosti smo prav tako uporabili vezje na sliki 5.16. Ko vzbujevalni signal preseže zgornjo ali spodnjo vrednost napajalne napetosti izhodni signal nima več sinusne oblike, s čimer se vrednost popačenja močno poveča. Tokrat nas zanima vrednost napajalne napetosti, ki jo moramo dovajati vezju,

da je vrednost popačenja vzbujevalnega signala amplitude 2 V_{rms} znotraj meja tolerance. Preverjanje smo začeli pri napajalni vrednosti ±3,328 V, ki je nekoliko večja od vrednosti amplitude signala po pretvorbi iz V_{rms} v V_p (2 V_{rms} * √2 = 2,828 V_p). Vrednost napajalne napetosti smo postopoma povečevali za korak ±0,3 V.



Slika 5.19 : Zajem karakteristike popačenja v odvisnosti od napajalne napetosti

Na sliki 5.19 vidimo, da so šele pri vrednosti napajalne vrednosti ±4,228 V vse višje harmonske komponente manjše od -80 dBc, kar je tudi bil iskani pogoj naše meritve.

6 Zaključek

Zasnovali in razvili smo C++ knjižnico, ki omogoča avtomatizirano uporabo treh laboratorijskih inštrumentov. Razvita knjižnica uporabniku skrije tehnične podrobnosti inštrumentov in njihove komunikacije z osebnim računalnikom, s čimer smo dosegli zadani cilj.

Na primeru razvitih demonstracijskih programov, smo prikazali prednosti uporabe razvite knjižnice pri ustvarjanju merilnih aplikacij. Programi skrajšajo čas zajema karakteristik obravnavanega vezja v primerjavi z ročnim izvajanjem meritev. Parametri in rezultati vseh meritev se vzporedno shranjujejo v tekstovne datoteke, tako da jih enostavno uvozimo v programe za nadaljnjo obdelavo.

Razvita C ++ knjižnica je tudi podlaga za razvoj C++ razredov ostalih laboratorijskih inštrumentov, ki ponujajo možnost povezave z osebnim računalnikom preko različnih vodil.

Literatura

[1] Hameg HM8118 Manual: http://www.hameg.com/manuals.0.html?&no_cache=1&L=0

[2] Hameg LCR Measurement Bridge HM8118: http://www.hameg.ru/438.0.html?&L=pxgufrzin

[3] Laboratory Power Supply QL355T & QL355TP: http://www.tti.eu/products-tti/text-pages/psu-ql-series.htm

[4] QL355T & QL355TP Instruction Manual, Thurlby Thandar Instruments: http://tti1.co.uk/downloads/manuals/QL355T%20&%20QL355TP%20Instruction%20Manual%20-%20Iss%207.pdf

[5] Model 2016-P Audio Analyzing DMM: http://www.keithley.com/products/dcac/dmm/application/?mn=2016-P

[6] Model 2016 THD Multimeter User's Manual: http://www.keithley.com/data?asset=847

[7] 6½-Digit THD Multimeters, 6½-Digit Audio Analyzing Multimeters: http://www.keithley.com/data?asset=363

[8] Programski jezik C++: http://slo-tech.com/clanki/04009/

[9] Bjarne Stroustrup, The C+ + Programming Language, Third Edition, AT&T Labs, Murray Hill, New Jersey, 1997

[10] Programming - Lecture slides: http://stroustrup.com/Programming/lecture-slides.html

[11] ASCII & HTML Codes : http://ascii.cl/references.htm

[12] Understand SINAD, ENOB, SNR, THD, THD + N and SFDR so You Don't Get Lost in the Noise Floor: http://www.analog.com/static/imported-files/tutorials/MT-003.pdf

[13] David M. Levine, Patricia P. Ramsey in Robert K. Smidt, Applied statistics for engineers and scientists: using Microsoft Excel and MINITAB, Prentice−Hall, Upper Saddle River, New Jersey, 2001

[14] P. Driscoll, F. Lecky, M.Crosby. 2000. An introduction to everyday statistics−2. Dostopno prek: http://emj.highwire.org/content/17/4/274.full

[15] Histogram: http://www.icoachmath.com/math_dictionary/Histogram.html

[16] Boštjan Murovec, Laboratorijske vaje Industrijska elektronika in elektronika z digitalno tehniko, 1. izdaja, Založba FE in FRI, Fakulteta za elektrotehniko, Ljubljana, 2004

[17] Orit Hazzan's Colum. Abstraction in Computer Science & Software Engineering: A Pedagogical Perspective.
Dostopno prek:
http://edu.technion.ac.il/Faculty/OritH/HomePage/FrontierColumns/OritHazzan_SystemDesigFrontier_Column5.pdf

Izjava

Izjavljam, da sem diplomsko delo izdelal samostojno pod vodstvom mentorja doc. dr. Boštjana Murovca, univ. dipl. inž. el. Izkazano pomoč drugih sodelavcev sem v celoti navedel v zahvali.

Marko Đorić