UNIVERZA V LJUBLJANI

FAKULTETA ZA MATEMATIKO IN FIZIKO

ODDELEK ZA FIZIKO

Zaključna naloga pri predmetu

UPORABA MIKROPROCESORJEV

Gregor Jecl

Ljubljana, september 2018

1 Naloga

Potrebujemo generator zašumljenega harmonskega signala nastavljive amplitude in frekvence, ki deluje med 10 Hz in 20 kHz. Frekvenca naj bo nastavljiva v korakih po 0.1 Hz, prav tako pa naj bo nastavljiva količina dodanega šuma. Frekvenca vzorčenja naj bo takšna, da bo v vsaki periodi generiranih najmanj 10 vzorcev. Parametri delovanja generatorja naj bodo izpisani na zaslonu mikroprocesorja in nastavljivi z gumbi na mikroprocesorski plošči.

2 Reševanje

Mikroprocesor STM32F406VG, ki smo ga uporabljali pri vajah in za reševanje naloge, mora generirati sinusni signal, katerega oblika se ne bo spreminjala, frekvenca pa mora biti nastavljiva. Zaradi teh dveh zahtev in zaradi preprostosti metode se odločimo, da bomo signal generirali v DDS tehniki. Prva ideja, kako naložiti šum na generirani sinusni signal je bila, da bi to naredil digitalno analogni pretvornik (DAC) sam. DAC je namreč mogoče konfigurirati tako, da vrednosti signala, preden ga iz DHR registra (*Data Holding Register*) prenese v DOR (*Data Output Register*), prišteje šum (naključno število), katerega amplitudo je mogoče spreminjati s spreminjanjem maske v LFSR registru (*Linear Feedback Shift Register*), ki služi kot generator pseudo naključnih števil. Šum na LFSH je mogoče spreminjati v korakih po bitih tj. 1 do 12 bitov šuma oziroma od amplitude šuma 1 do amplitude šuma 4095. Vendar je tako generirani šum bel, kar pomeni, da je njegova spektralna gostota konstantna, v nalogi pa želimo Gaussovski šum, kar je šum, kot ga dobimo naprimer na uporniku. Iz belega lahko do Gaussovskega šuma elegantno pridemo tako, da povprečujemo zaporedne vrednosti belega šuma (gre za uporabo centralnega limitnega izreka, ki pravi, da ima vsote velikega števila neodvisnih dogodkov Gaussovsko porazdelitev četudi verjetnostne porazdelitve teh dogodkov niso Gaussovska), vendar žal DAC takega povprečevanja vrednosti šuma ne omogoča. Na tej točki obstajata dve možnosti: generiranje naključnih števil v softwaru ali uporaba v STM32F407VG vgrajenega generatorja naključnih števil (*RNG - Random Number Generator*). Odločimo se za drugo možnost, ker je bolj zanimiva.
Plan je torej sledeč: v prekinitvenih rutinah s pomočjo DDS tehnike in DACa generiramo sinusni signal, ki mu prištejemo povprečje nekaj (v izhodiščnem primeru 32) vrednosti, ki jih RNG generira v svojih prekinitvenih rutinah.
V nadaljevanju najprej sledi opis pomembnejših delov rešitve oziroma uporabljenih tehnik, nato koda in razlaga posameznih delov programa in izbranih opcij, na koncu pa še ovrednotenje delovanja.

2.1 DDS tehnika

Pri DDS tehniki gre za to, da vrednosti, ki jih generiramo, ne računamo sproti, temveč jih s pomočjo nekega kazalca beremo iz v naprej pripravljene tabele, ki mora vsebovati primerno število vzorcev ene periode sinusnega signala. Tehniko je sicer mogoče uporabiti tudi za druge oblike signalov, kjer moramo spreminjati le frekvenco in amplitudo signala, oblike pa ne. Na začetku izvajanja glavne zanke programa najprej inicializiramo tabelo in vanjo shranimo primerno število vrednosti, v našem primeru 4096. Vsakič, ko timer sproži prekinitev, na DAC pošljemo vrednost iz tabele in inkrementiramo kazalec. Frekvenca generiranega signala se spremeni, če iz tabele ne beremo zaporednih vrednosti, temveč vrednosti, razmaknjene za nek K. V takem primeru torej (ker prekinitve prihajajo v enakomernih časovnih intervalih) tabelo preberemo "hitreje", torej z višjo frekvenco. Paziti moramo, da kazalec ne pade iz tabele, zato ga omejimo z *and*. Kazalec tako vedno kaže na enakomerno razmaknjena mesta v tabeli in ni

važno, ali je velikost (vrednost) K večkratnik dolžine tabele, saj v naslednjem obhodu kazalec pač pobere druge vrednosti, važno je le, da je razmak med njimi konstanten [1, 2].
Frekvenca tako generiranega signala je

$$f = f_0' \frac{K}{2^n}, \tag{1}$$

kjer je f_0' efektivna frekvenca prekinitev tj. frekvenca timerja f_0 podeljena s *prescalerjem* (zato pridevnik efektivna), n število bitov števca v timerju in K je K, torej parameter, s spreminjanjem katerega spreminjamo frekvenco generiranega signala. V našem primeru je $f_0 = 250$ kHz (ker mora biti tudi pri najvišji zahtevani frekvenci, 20 kHz, v eni periodi najmanj 10 vrednosti, pa še malo rezerve), $n = 16$ (tako pač je), prescaler pa uporabimo za povečanje občutljivosti nastavljanja frekvence (več v nadaljevanju).

2.2 Šum in generator naključnih števil

Generator naključnih števil RNG je vgrajena periferna enota, ki daje uporabniku na voljo 32-bitna pseudo naključna števila z LFSR, ki ga incializirajo "naključna semena" z analognega vezja oscilatorjev [3]. RNG poda novo vrednost vsakih 40 period ure PLL48CLK, ki dela pri 48 Mhz, torej vsakih 0.8 μs [4]. Izkaže se torej, da bo naš šum, vsaj pri visokih frekvencah generiranega signala, koreliran, saj v času med dvema prekinitvama (~ 4 μs) dobimo le 4 nova naključna števila. Namesto 32 vrednosti šuma zato raje povprečujemo 16 vrednosti, ker na tak način zmanjšamo korelacijo v šumu ter pospešimo izvajanje "glavne" prekinitvene rutine (tiste, v kateri pošiljamo vrednosti na DAC), čeprav malo pokvarimo "Gaussovskost".
V programu implementiramo dodajanje šuma tako, da v glavni zanki preverjamo stanje gumbov na mikroprocesorski plošči (za večanje oziroma manjšanje amplitude šuma je treba pritisniti dva gumba hkrati), v vsaki glavni prekinitvi povprečimo 16 naključnih vrednosti, shranjenih v tabeli, nato pa dobljeno vrednost prištejemo vrednosti, ki gre na DAC na račun generiranja z DDS.

2.3 Nastavljivost frekvence

Po enačbi (1) izračunamo, da je frekvenca generatorja v primeru, da je prescaler enak 1 in se K spremeni za 1, nastavljiva na ~ 3.8 Hz, kar je nad zahtevano vrednostjo $\Delta f = 0.1$ Hz. Pri tako visoki frekvenci vzorčenja in pri $n = 16$ bitnih števcih je nemogoče doseči zahtevani Δf, če ne spremenimo vrednosti prescalerja. Prescaler pove, kolikokrat mora timer prešteti do $2^{16} = 65536$, preden sme sprožiti prekinitev. Z večanjem vrednosti prescalerja manjšamo efektivno frekvenco prekinitev kar omogoča finejšo nastavljanje frekvence.
Zahtevi po $\Delta f = 0.1$ Hz zaradi Nyquistovega kriterija ne moremo ugoditi po celem frekvenčnem območju. Naj bo vrednost prescalerja 40. S tem, ko efektivno frekvenco zmanjšamo s 250 kHz na $250/40 = 6.2$ kHz smo, skupaj s pogojem glede 10 vrednosti na vsako periodo, omejili najvišjo frekvenco, ki jo še lahko zadovoljivo generiramo na ~ 620 Hz (če uporabimo iste meje za K, kot v primeru, da je prescaler enak 1, je mogoče generiranje frekvence največ do 20 kHz$/40 = 500$ Hz). Uporabniku je torej treba dati možnost, da izbira, kako natančno želi generirati frekvenco. V programu smo to implementirali tako, da pritisk modrega gumba na mikroprocesorski plošči (gumb imenovan B1, na portu A0) sproži prekinitev, v kateri nastavimo bit TIM5_PSC na 39 oziroma ob ponovnem pritisku na 0, saj velja prescaler = TIM5_PSC + 1.
Prvotna ideja je sicer bila, da bi se vrednost prescalerja samodejno spreminjala v korakih

$40, 4, 1$ s tem, ko bi uporabnik spreminjal frekvenco, saj bi bilo na tak način mogoče vzdrževati primernejšo (višjo) občutljivost v posameznih frekvenčnih območjih, v naši implementaciji pa ima uporabnik samo dve možnosti. Izkaže pa se, da taka rešitev ne bi bila dobra, ker morje *if* stavkov, ki jih je potrebno za to napisati, upočasnijo glavno zanko programa do te mere, da program več ne deluje, saj je glavna zanka počasnejša od prekinitev, pride pa tudi do problemov z vrednostmi K, če želimo, da se frekvenca spreminja zvezno (izkušenejši programerji bi morda imeli manj težav).

3 Koda programa

Program začnemo s primernimi *include*-i. Poleg običajnih ("stm32f4xx.h", "stm32f4xx_rcc.c", "dd.h") potrebujemo tudi *source* datoteke za ostale periferne enote, ki jih bomo uporabljali.

```
1  #include "stm32f4xx.h"
2  #include "stm32f4xx_rcc.c"
3  #include "stm32f4xx_rng.c"
4  #include "stm32f4xx_dac.c"
5  #include "stm32f4xx_gpio.c"
6  #include "stm32f4xx_tim.c"
7  #include "stm32f4xx_exti.c"
8  #include "stm32f4xx_syscfg.c"
9  #include "dd.h"
10 #include "LCD2x16.c"
11 #include "math.h"
```

Enote, ki jih uporabljamo so: RNG ("stm32f4xx_rng.c"), DAC ("stm32f4xx_dac.c"), porti ("stm32f4xx_gpio.c"), timer TIM5 ("stm32f4xx_tim.c"), LCD prikazovalnik ("LCD2x16.c"), datoteki "stm32f4xx_exti.c" in "stm32f4xx_syscfg.c" pa potrebujemo za prekinitve na gumbu B1.

```
12 int Buffer[16], DDSTable[4096], ptrTable, ptrBuffer = 0;
13 int k = 262, Am = 150, AmNoise = 0, prs = 1.0;
```

V tej vrstici inicializiramo tabele in spremenljivke, ki jih bomo kasneje uporabljaliv prekinitvenih rutinah (definirati jih moramo kot globalne). Tabela *Buffer* je tabela, v katero bo RNG pisal naključne vrednosti, ki jih bomo povprečevali. Tabela *DDSTable* bo vsebovala vrednosti ene periode sinusnega signala. *ptrTable*, *ptrBuffer* sta kazalca v prej omenjeni tabeli, $k = K$, Am je relativna amplituda generiranega sinusnega signala ($Am \in (0, 255)$, celotna amplituda bo nekje med 0 in 1850, kjer 2048 ustreza sredini območja delovanja 12-bitnega DACa, ki je zmožen generirati napetosti med 0 in 3 V), *AmNoise* je relativna amplituda dodanega šuma, *prs* pa je vrednost prescalerja, ki bo lahko 1 ali 40.

```
14 //Initilialize Button B1 - PA0
15 void GPIOAinit (void)   {
16 GPIO_InitTypeDef  GPIO_InitStructure;
17   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
18   GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_0;
```

```
19    GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_IN;
20    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;
21    GPIO_Init(GPIOA, &GPIO_InitStructure);
22  }
23
24  // initialize port E, Gpio_Pin_8 to GPIO_Pin_15 as outputs
25  void GPIOEinit (void)   {
26  GPIO_InitTypeDef  GPIO_InitStructure;
27    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);
28    GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_8;
29    GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_OUT;
30    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
31    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
32    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;
33    GPIO_Init(GPIOE, &GPIO_InitStructure);
34  }
```

V vrsticah 15-22 kot vhod konfiguriramo gumb B1, ki se nahaja na pinu 0 porta A. V vrsticah 25-33 konfiguriramo pin 8 na portu E kot izhod [5]. Pin 8 uporabljamo za merjenje časa trajanja glavne prekinitvene rutine (ali katere druge, ali glavne zanke v programu) z osciloskopom.

```
35  void SWITCHinit (void)  {
36  GPIO_InitTypeDef        GPIO_InitStructure;
37    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);
38    GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_3 | GPIO_Pin_4 |
39                                    GPIO_Pin_5 | GPIO_Pin_6;
40    GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_IN;
41    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
42    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
43    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_DOWN;
44    GPIO_Init(GPIOE, &GPIO_InitStructure);
45  }
```

Na tej točki konfiguriramo pine 3, 4, 5 in 6 na portu E, kjer se nahajajo gumbi, s katerimi bomo spreminjali frekvenco, amplitudo in amplitudo šuma. Konfiguracija je vzeta iz [5].

```
46  // DAC init function
47  void DACinit (void)     {
48  DAC_InitTypeDef         DAC_InitStructure;
49  GPIO_InitTypeDef        GPIO_InitStructure;
50    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
51    GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_4 | GPIO_Pin_5;
52    GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AN;
53    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;
54    GPIO_Init(GPIOA, &GPIO_InitStructure);
55
56    RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);
57    DAC_InitStructure.DAC_OutputBuffer                = DAC_OutputBuffer_Enable;
58    DAC_InitStructure.DAC_Trigger                     = DAC_Trigger_None;
59    DAC_InitStructure.DAC_WaveGeneration              = DAC_WaveGeneration_None;
60    DAC_Init(DAC_Channel_1, \&DAC_InitStructure);
```

```
61    DAC_Init(DAC_Channel_2, \&DAC_InitStructure);
62
63    DAC_Cmd(DAC_Channel_1, ENABLE);
64    DAC_Cmd(DAC_Channel_2, ENABLE);
65  }
```

Sledi konfiguriranje DACa skupaj s pinoma 4 in 5 porta A, ki ju konfiguriramo kot analogna (GPIO_Mode_AN), saj bosta to izhoda iz mikroprocesorja, na katerih bomo z osciloskopom opazovali signal. Konfiguriramo oba DACa, ker zakaj pa ne, čeprav bomo šum nalagali le na enega. DACov ne bomo prožili z zunanjim signalom, nočemo dodatnega trikotnega signala ali šuma, uporabimo pa buffer, da dobimo več toka na izhodih [6].

```
66  // RNG init function
67  void RNGinit(void)      {
68    RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_RNG, ENABLE);
69
70    NVIC_EnableIRQ(HASH_RNG_IRQn);             // Enable RNG IRQ in NVIC
71    RNG->CR = RNG_CR_IE | RNG_CR_RNGEN;  // Enable RNG IRQ or enable RNG
72  }
```

V naslednjem koraku konfiguriramo RNG, kar je preprosteje kot pri ostalih perifernih enotah, saj moramo le omogočiti uro. V vrstici 70 povemo enoti NVIC, ki nadzoruje prekinitve, da želimo omogočiti prekinitev z RNG in sicer preko prekinitvene rutine HASH_RNG_IRQn (ima prioriteto 80), v vrstici 71 pa v kontrolnem registru (*Control Register*) enote RNG postavimo bita RNG_CR_IE (*Interrupt Enable*) in RNG_CR_RNGEN (*Random Number Generation*), kjer prvi omogoči RNGju, da zahteva prekinitve, drugi pa požene enoto (*ENABLE* bit).

```
73  // Timer 5 init function - time base
74  void TIM5init_TimeBase_ReloadIRQ (int interval)  {
75  TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure;
76    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5,  ENABLE);
77    TIM_TimeBaseInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;
78    TIM_TimeBaseInitStructure.TIM_CounterMode = TIM_CounterMode_Up;
79    TIM_TimeBaseInitStructure.TIM_Period = interval;
80    TIM_TimeBaseInitStructure.TIM_Prescaler = 0;
81    TIM_TimeBaseInitStructure.TIM_RepetitionCounter = 0;
82    TIM_TimeBaseInit(TIM5, &TIM_TimeBaseInitStructure);
83
84    NVIC_EnableIRQ(TIM5_IRQn);                 // Enable IRQ for TIM5 in NVIC
85    TIM_ITConfig(TIM5, TIM_IT_Update, ENABLE); // Enable IRQ on update for Timer5
86
87    TIM_Cmd(TIM5, ENABLE);
88  }
```

Timer konfiguriramo (po [7]) tako, da periodično, ob vnovičnem začetku štetja, pošilja zahteve za prekinitev na NVIC.

```
89  // IRQ Button B1 - PA0 interupt config
90  void IRQinit_EXTI (void) {
91  EXTI_InitTypeDef        EXTI_InitStructure;
92
```

```
93    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);
94
95    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0);
96
97    EXTI_InitStructure.EXTI_Line   |= EXTI_Line0;
98    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
99    EXTI_InitStructure.EXTI_Mode    = EXTI_Mode_Interrupt;
100   EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
101   EXTI_Init(&EXTI_InitStructure);
102
103   NVIC_EnableIRQ(EXTI0_IRQn);             // Enable IRQ for ext. signals, line 0
104   NVIC_SetPriority(EXTI0_IRQn, 51);       // Must have lower priority than DAC
105 }
```

Zadnji korak pred funkcijo *main* je konfiguracija EXTI kontrolerja (*External Interrupt Controller*), ki poskrbi, da s pritiskom na gumb, ki smo ga konfigurirali v vrsticah 15-21, sprožimo prekinitev. Da EXTI ve, od kod bo prišel signal, odgovoren za prekinitev, je treba konfigurirati multiplekserje, ki iz 144 pinov na mikroprocesorski plošči izberejo pravega (pin 0 na portu A - vrstica 95) - to storimo v bloku SYSCFG [8]. V vrstici 104, potem, ko smo NVICu v vrstici 103 povedali, da se naj odziva na prekinitve z EXTI0, nastavimo prioriteto te prekinitve na 51. Razlog je v tem, da imajo prekinitve s timerja TIM5 prioriteto 50, mi pa želimo, da generiranje signala potega čim bolj nemoteno.

Sledi glavni del programa: funkcija main().

```
106 int main ()  {
107
108   int sw, Fp;
109   for (ptrTable = 0; ptrTable <= 4095; ptrTable++)
110     DDSTable[ptrTable] = (int)(1850.0 * sin((float)ptrTable / 2048.0 * 3.14159265));
111
112   GPIOAinit();
113   GPIOEinit();
114   SWITCHinit();
115   LCD_init();
116   LCD_string("Frq=", 0x00); LCD_string("Hz", 0x0e); LCD_string("Am,Ns=", 0x40);
117   DACinit();
118   RNGinit();
119   IRQinit_EXTI();
120   TIM5init_TimeBase_ReloadIRQ(336);          // 336 == 4 us == 250 kHz
121
122   while (1) {                                // endless loop
123     sw = GPIOE->IDR;
124
125     if ((sw & S370) && (sw & S371) && (AmNoise > 0)) AmNoise--;
126     if ((sw & S372) && (sw & S373) && (AmNoise < 255)) AmNoise++;
127     if ((sw & S371) && (k  < 5243)) k++;       // Change frequency
128     if ((sw & S370) && (k  >     2)) k--;       //
129     if ((sw & S373) && (Am <   255)) Am++;      // Change signal amplitude
130     if ((sw & S372) && (Am >     1)) Am--;      //
131
```

```
132     Fp = (int)(250.0e3 * (float)k / (prs * 65535.9));     // Frequency on LCD
133     LCD_uInt16(Fp, 0x09, 1);
134     LCD_uInt16(Am, 0x46, 1);
135     LCD_uInt16(AmNoise, 0x4b, 1);
136   };
137 }
```

V vrstici 108 inicializiramo naravni števili *sw* in *Fp*. Prva je stanje gumbov, ki ga beremo v vrstici 123, druga pa je frekvenca, ki se bo izpisovala na zaslon LCD prikazovalnika. Tabelo za DDS generiramo v vrsticah 109 in 110, v vrsticah 112 do 120 pa pokličemo vse konfiguracijske funkcije ter zaženemo periferne enote. Vrstica 116 tukaj izstopa, saj v njej ne inicializiramo enot, temveč na LCD v prvo vrstico zapišemo tiste znake, ki se med delovanjem ne bodo spreminjali: frekvenca v enotah herzov, amplituda, šum (noise). V vrstici 120, v kateri inicializiramo timer TIM5 in njegove prekinitve, opazimo, da je argument funkcije 336, kar ustreza štirim mikrosekundam med dvema dvema overflow-oma timerja oziroma frekvenci 250 kHz (če je prescaler enak 1).
Sledi neskončna zanka, v kateri najprej preberemo stanje gumbov, nato pa šest *if* stavkov, ki primerjajo stanja gumbov in skladno s tem večajo amplitudo šuma (vrstici 125 in 126 - naenkrat moramo pritisniti dva gumba), frekvenco signala (vrstici 127 in 128) in amplitudo signala (129 in 130). Opazimo, da je števec k v vrsticah 127 in 128 omejen: omejitvi ustrezata frekvencam $\sim$ 8 Hz in 20 kHz (če je prescaler enak 1). Tudi amplituda ima svoje omejitve, vendar je tu številka 255 bolj arbitrarna: amplitudo želimo spreminjati v dovolj finih koraki, vendar nočemo, da bi bili koraki premajhni, kot bi bili, če bi vzeli kako večje število (iz [2]). Amplituda šuma je omejena iz istega razloga na enak način.
V vrstici 132 izračunamo frekvenco, ki se bo izpisovala na prikazovalniku, v vrstici 133 jo na le-tega zapišemo, v vrsticah 134 in 135 pa zapišemo še amplitudo sinusa in amplitudo nanj naloženega šuma.

V nadaljevanju sledijo prekinitvene rutine:

```
138 // IRQ function for Timer5
139 void TIM5_IRQHandler(void)       {
140   GPIOE->BSRRL = BIT_8;
141
142   TIM_ClearITPendingBit(TIM5, TIM_IT_Update); // clear interrupt flag
143
144   int noise = 0;
145   for(int i = 0; i < 16; i++) {         // Sum all values in Buffer for averaging
146     noise += (Buffer[i & 15]);
147   }
148
149   ptrTable = (ptrTable + k) & 0xffff;
150   int Out1 = ((Am * DDSTable[  ptrTable >> 4]) / 256
151               + 2048 + ((AmNoise * ((noise - 32768) / 16)) / 256));
152               // Substract 16*2048 from noise and divide by 16
153               // to get Gaussian noise N(0, sigma)
154
155   if (Out1 > 4095) Out1 = 4095;
156   if (Out1 < 0) Out1 = 0;
157
158   DAC->DHR12R1 = Out1;
```

```
159   DAC->DHR12R2 = (Am * DDSTable[((ptrTable >> 4) + 1024) & 4095]) / 256 + 2048;
160
161   GPIOE->BSRRH = BIT_8;
162 }
```

Najpomembnejši del programa je ravno prekinitvena rutina TIM5_IRQHandler, ki temelji na [2]. Bit 8, ki ga postavimo in podremo v vrsticah 140 in 161 služi merjenju časa izvajanja. V vrstici 142 najprej podremo prekinitveno zastavico, da NVIC ne skoči takoj ponovno na začetek prekinitve. Vrstica 144 inicializira naravno število *noise*, v katerega v *for* zanki v vrsticah 145-147 seštejemo vse vrednosti šuma, ki jih nameravamo poslati iz DACa. Vrstica 149 inkrementira in omeji kazalec v DDS tabelo, vrstici 150 in 151 pa inicializirata pomožno spremenljivo, ki bo pravzaprav output iz DACa.
Pomožna spremenljivka je vsota vrednosti iz DDS tabele, ki je pomnožimo s številom $\in (0, 1)$, s katerim definiramo amplitudo ($Am/256$), števila 2048, ki predstavlja polovico območja DACa (da dobimo signal na sredi območja) in šumnega člena. Šumni člen sestavlja prej dobljena vrednost *noise*, od katere najprej odštevejmo 16×2048, ter podelimo s 16. To storimo zato, da šum, ki ga prištejemo členu iz DAC, vedno pleše okrog vrednosti 2048 oziroma zato, da pri povečevanju amplitude šuma ne večamo skupne "količine" signala, saj bi v tem primeru na neki točki zadeli najvišjo možno napetost, ki jo je DAC še sposoben dati in bi porezali signal. Za povečevanje oziroma zmanjševanje količine šuma, podobno kot pri členu z DACa, pomnožimo vrednost z $AmNoise/256$.
V vrsticah 155 in 156 omejimo vrednost pomožne spremenljivke, da na osciloskopu, če gredo vrednosti preko zmogljivosti DAC, ne dobimo čudnih pojavov. Vse skupaj nato v vrstici 158 pošljemo na DHR register DACa, v naslednji vrstici pa na drugi DAC pošljemo 90 stopinj zakasnjeni, nezašumljeni signal.

```
163 // IRQ function for RNG
164 void HASH_RNG_IRQHandler(void)  {
165   Buffer[(ptrBuffer++ & 15)] = (RNG->DR & 0xfff);
166   // Increment pointer and store 12-bits of random number
167 }
```

Prekinitvena rutina, ki jo kliče RNG je preprosta, v njej inkrementiramo in omejimo kazalec ter v tabelo Buffer zapišemo novo naključno vrednost, omejeno na 12-bitov (ker je DAC 12-bitni).

```
168 // IRQ function for Button B1 - PA0
169 void EXTI0_IRQHandler (void)   {
170   if (prs == 1){
171     TIM5->PSC = 39;
172     prs = 40;
173   }
174   else {
175     TIM5->PSC = 0;
176     prs = 1;
177   }
178   EXTI_ClearITPendingBit(EXTI_Line0);
179   EXTI_ClearFlag(EXTI_Line0);
180 }
```

Zadnja stvar v programu je prekinitvena rutina, ki jo sprožimo s pritiskom na gumb B1, torej

rutina, ki spreminja prescaler. V ta namen je uporabljena le preprosta *if* zanka. V vrsticah 178 in 179 pa je potrebno še podreti zastvice, odgovorne za prekinitev.
S tem se zaključi obravnava in komentiranje kode programa.

4 Delovanje

4.1 Šum

Šum, ki ga ob vsaki prekinitvi naložimo na DAC, pokriva celotno območje DACa, saj lahko ima vrednosti od 0 do 4096, vendar verjetnosti za posamezne vrednosti zaradi povprečevanja niso več enake. Šum se ob pritiskanju na gumbe mikroprocesorske plošče spreminja ažurno, saj program spremembo zazna ob v vsakem obhodu glavne zanke. Če z osciloskopom pogledamo Fourierovo transformacijo signala pa vidimo, da čeprav šum ni bel, tudi Gaussovski ni. Pri večkratnikih 62.5 kHz opazimo, da ima spektralna gostota šuma, ki sicer pada s frekvenco, ničle, katerih položaj je neodvisen od frekvence generiranega signala. Dejstvo, da je 62.5 kHz točno 1/4 frekvence timerja daje misliti, da je pojav povezan z le-to in da smo priča frekvenčni karakteristiki nekega filtra.
Po razmisleku, ki je trajal dlje, kot bi bilo primerno, se je avtor spomnil, da je bločno povprečevanje proces, ki ga v digitalni elektroniki popišemo s FIR filtrom s frekvenčno karakteristiko [1]

$$T(i\omega) = \frac{\sin\left(2\pi\frac{f_0'}{f_b}\right)}{2\pi\frac{f_0'}{f_b}}, \tag{2}$$

kjer je f_b "frekvenca povprečevanja". Vemo, da so ničle pri frekvenci $f_{nicle} = f_0'/4 = 62.5$ kHz, zato $2\pi\frac{f_0'/4}{f_b} = \pi$ in tedaj $f_b = f_0'/2$.
Čeprav lahko trdimo, da je dobljena frekvenčni spekter rezultat povprečevanja, vseeno ni čisto jasno, zakaj dobimo za frekvenco povprečevanja polovico frekvence timerja in so ničle spektra pri frekvencah 62.5 k kHz, kjer $k \in \mathbb{Z}$. Glede na število vrednosti, ki jih povprečujemo, bi morali dobiti $f_b = f_0'/8$, saj povprečujemo 16 vrednosti. Zgornje bi lahko morda pripisali dejstvu, da RNG v času med dvema prekinitvama uspe zgenerirati le 4 nove vrednosti (razdelek 2.2) namesto 16 in se zato efektivna frekvenca povprečevanja poveča za štirikrat. Razlaga se zdi verjetna predvsem zato, ker so numerične vrednosti med posameznimi količinami ravno prave, avtorja pa vendarle preseneča dejstvo, da bi filter, čeprav se v vsaki prekinitveni rutini sešteje vseh 16 števil, iz katerih se kasneje tvori povprečje, "vedel", katere vrednosti so "nove".

4.2 Amplituda in frekvenca

Amplituda in frekvenca se, kakor šum, spreminjata ažurno. Če se omejimo na delovanje v primeru, ko je prescaler enak 1, večjih presenečenj ni. Frekvenca se spreminja od ~ 8 Hz do 20 kHz, amplitudo pa je mogoče nastavljati na 3 V/256 = 0.012 V natančno.
V načinu občutljivega spreminjanja frekvence pa pride do dveh novih pojavov. Prvi je dejstvo, da je najvišja frekvenca, ki jo je še mogoče generirati v tem načinu, 500 Hz, saj meje za K ostanejo enake in velja 20 kHz/40 = 500 Hz. Signal, generiran pri tej frekvenci, pa je vse prej kot lep, saj se v eni periodi generira le 12.5 vzorcev, kar je za človeško oko premalo.
Do drugega pojava pa pride, ko gremo s frekvenco proti spodnji limiti, saj pri $\sim 6 - 7$ Hz začne amplituda signala padati. Očitno nek del vezja deluje kot high pass filter s prelomno frekvenco v tem območju. Možen razlog bi lahko bil parazitska kapacitanca na izhodnem pinu mikrokontrolerja.

4.3 Prekinitvene rutine

Čas izvajanja glavne prekinitvene rutine je odvisen od števila naključnih vrednosti, ki jih povprečujemo. Pri povprečevanju 32 vrednosti je prekinitvena rutina trajala $\sim 2.5\mu$s, pri povprečevanju 16 vrednosti pa polovico, tj. $\sim 1.25\mu$s. Omejujoči faktor v programu tako ni bila prekinitvena rutina, temveč RNG, ki daje nove vrednosti relativno počasi.
Računanje vrednosti pomožne spremenljive $Out11$ traja približno 250ns. Dolg čas izvajanja lahko pripišemo prevsem šumnemu členu, kjer je treba odšteti, deliti in množiti. Nekaj nanosekund verjetno prinese tudi preverjanje velikosti signala in omejevanje, v primeru da je potrebno, vendar smo nekaj časa pripravljeni žrtvovati za lepši signal.
Časa izvajanja ostalih prekinitvenih rutin nismo merili. Čas izvajanja rutine za RNG bi moral biti proti izvajanju glavne rutine zanemarljiv, ker obsega le logične operacije in pa prepisovanje v RNG_DR register. Vendar pa se avtor, čeprav se rutina za spremembo prescalerja sproži le redko, na uporabnikov ukaz, in se vrednosti bita TIM5_PSC spremeni tako hitro, da je sprememba v delovanju DACa človeškim očem nezaznavna, na tej točki sprašuje, zakaj trajanja te spremembe ni pomeril.

5 Zaključek in izboljšave

Poraja se vprašanje, ali bi bilo mogoče delovanje programa izboljšati tako, da bi naključna števila generirali v softwareu. Glede na to, da ne potrebujemo 32-bitnih naključnih števil, kot jih je zmožen generirati v mikroprocesor vgrajeni RNG, bi se morda izkazalo, da je operiranje s 16-bitnimi short integerji hitrejše in bi na ta račun dobili kvalitetnejši šum.
Vprašljivo je tudi, ali je način, na katerega smo se lotili povečevanja nastavljivosti frekvence, dober oziroma smiselen. Verjetno bi bilo bolje, da bi mikrokontroler povezali z računalnikom in nato navodila glede spreminjanja prescalerja pošiljali npr. preko USART vodila. Na ta način bi lahko celo frekvenčno območje elegantno razdelili na več delov in bi bile vrednosti prescalerja izbrane bolj na gosto, prav tako pa ne bi imeli problema s tem, da nam na mikroprocesorski plošči zmanjkuje gumbov. Program v trenutni različici pa bi verjetno hitro optimirali tako, da bi izbrali programerja z več izkušnjami.

Literatura

[1] D. Ponikvar. Uporaba mikroprocesorjev. `https://www.fmf.uni-lj.si/~ponikvar/PDFji/Uporaba%20mikroprocesorjev%208.pdf`, Ljubljana, Avgust 2010. [Online; citirano 5. september, 2018].

[2] D. Ponikvar. Signal generation using DDS. `https://www.fmf.uni-lj.si/~ponikvar/STM32F407%20project/Ch16%20-%20Signal%20generation%20using%20DDS.pdf`, Unknown. [Online; citirano 7. september, 2018].

[3] STMicroelectronics. RM0090 - Reference Manual - STM32F405xx/07xx, STM32F415xx/17xx, STM32F42xxx and STM32F43xxx advanced ARM-based 32-bit MCUs. `https://www.fmf.uni-lj.si/~ponikvar/PDFji/DM00031020%20RM0090%20Reference%20Manual.pdf`, September 2013. [Online; citirano 7. september, 2018].

[4] STMicroelectronics. ANN3988 Application Note - Clock configuration tool for STM32F40xx/41xx/427x/437x microcontrollers. `https://www.st.com/content/ccc/resource/technical/document/application_note/2a/3d/ea/2a/97/4a/4a/96/`

DM00039457.pdf/files/DM00039457.pdf/jcr:content/translations/en.DM00039457.pdf, August 2013. [Online; citirano 7. september, 2018].

[5] D. Ponikvar. The use of ports. https://www.fmf.uni-lj.si/~ponikvar/STM32F407%20project/Ch4%20-%20The%20use%20of%20ports.pdf, Unknown. [Online; citirano 9. september, 2018].

[6] D. Ponikvar. Digital to analog converter. https://www.fmf.uni-lj.si/~ponikvar/STM32F407%20project/Ch7%20-%20Digital%20to%20analog%20converter.pdf, Unknown. [Online; citirano 9. september, 2018].

[7] D. Ponikvar. Interrupts and Timer. https://www.fmf.uni-lj.si/~ponikvar/STM32F407%20project/Ch11%20-%20Interrupts%20and%20Timer.pdf, Unknown. [Online; citirano 9. september, 2018].

[8] D. Ponikvar. Interrupts and ports. https://www.fmf.uni-lj.si/~ponikvar/STM32F407%20project/Ch10%20-%20Interrupts%20and%20ports.pdf, Unknown. [Online; citirano 9. september, 2018].