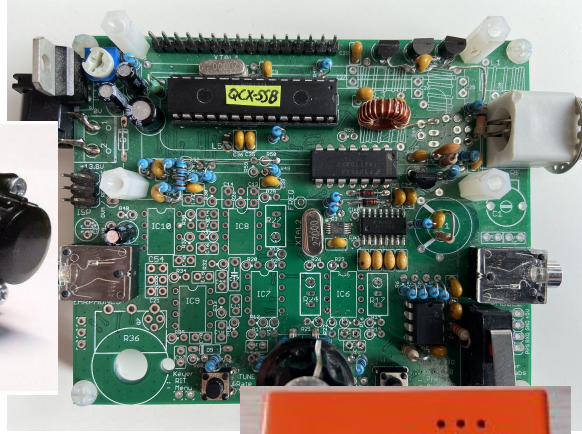
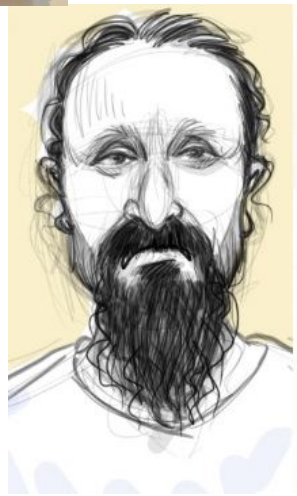


uSDX Anatomisi

Moderator : TA1DE Volkan SAMLI

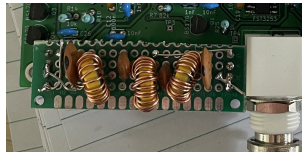


Baris Dinc
TA7W / OH2UDS

Konuk : WB2CBA Barbaros Asuroglu

Nedir uSDX ?

- SDR QRP Telsiz
- 0 - 100 Mhz (HF + 6m)
- All Mode (AM/FM/SSB/CW + Digital)
- 2.4 Khz (~3.8 KHz BW)
- DSP Filtre (50/100/200/500/1700/2500/4000 Hz)
- ~5 Watt PEP SSB (Class-E)
- VOX, AGC, ATT, NR, MORSE ENCODE/DECODE
- Atmega 328p (Arduino Uno)
- -135dBm dinleme hassasiyeti (28MHz, 200 Hz BW)
- ATT (zayiflatici) 0..-73dB
- CAT Destegi



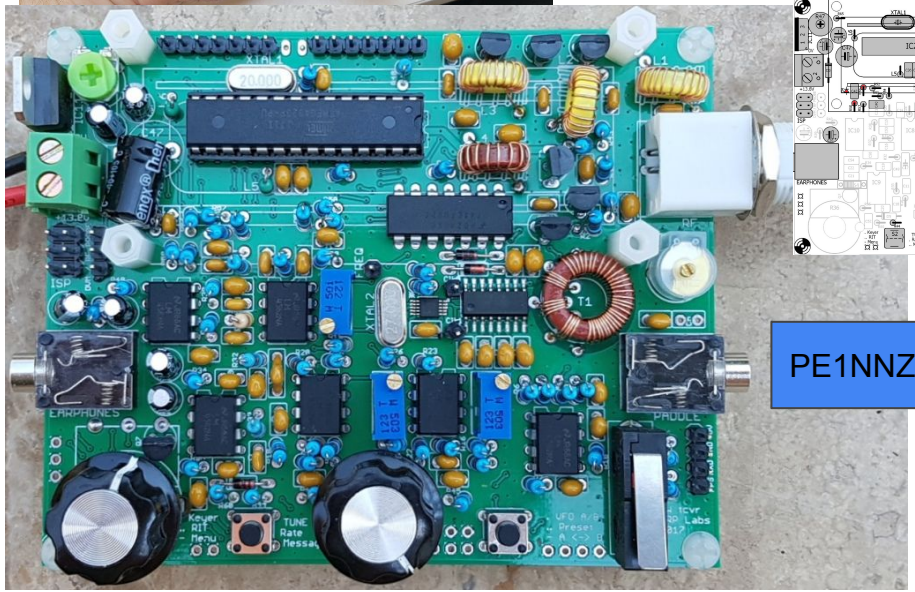
Hikayesi

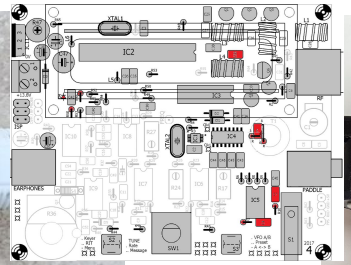
QRP Labs

DC transceiver;
NorCal 2030 by *Dan Tayloe (N7VE 2004)*
Hi-Per-Mite Active Audio CW Filter *David Cripe (NMØS)*,
Low Pass Filters *Ed (W3NQN 1983)*,
a key-shaping circuit by *Donald Huff (W6JL)*,
BS170 CMOS driven ATS MOSFET PA *Steven Weber (KD1JV)*
Ghetto-class-E filter-network *Paul Harden (NA5N)*
ATMEGA328P + HD44780 LCD + SI5351
Software Raspberry Pi digital SSB *Guido (PE1NNZ)*

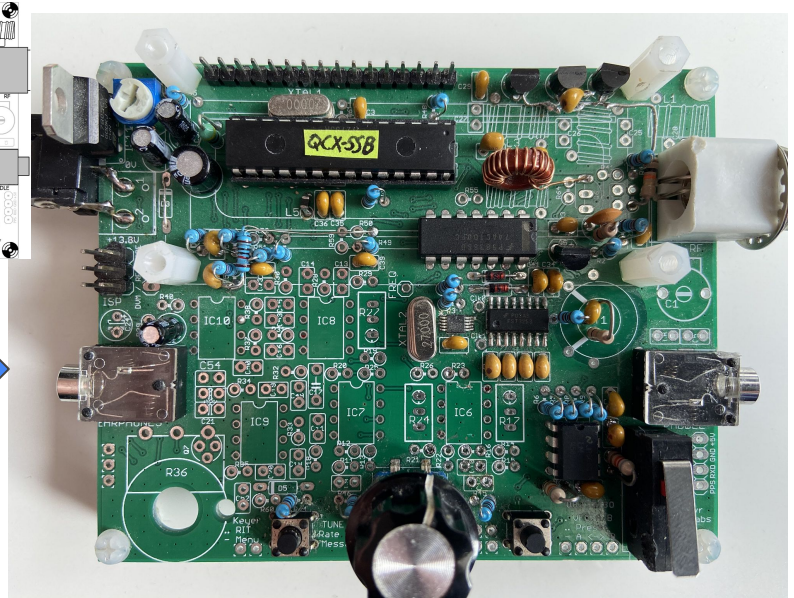
Hans Summers (G0UPL)
RSGB YOTA summer camp 2017

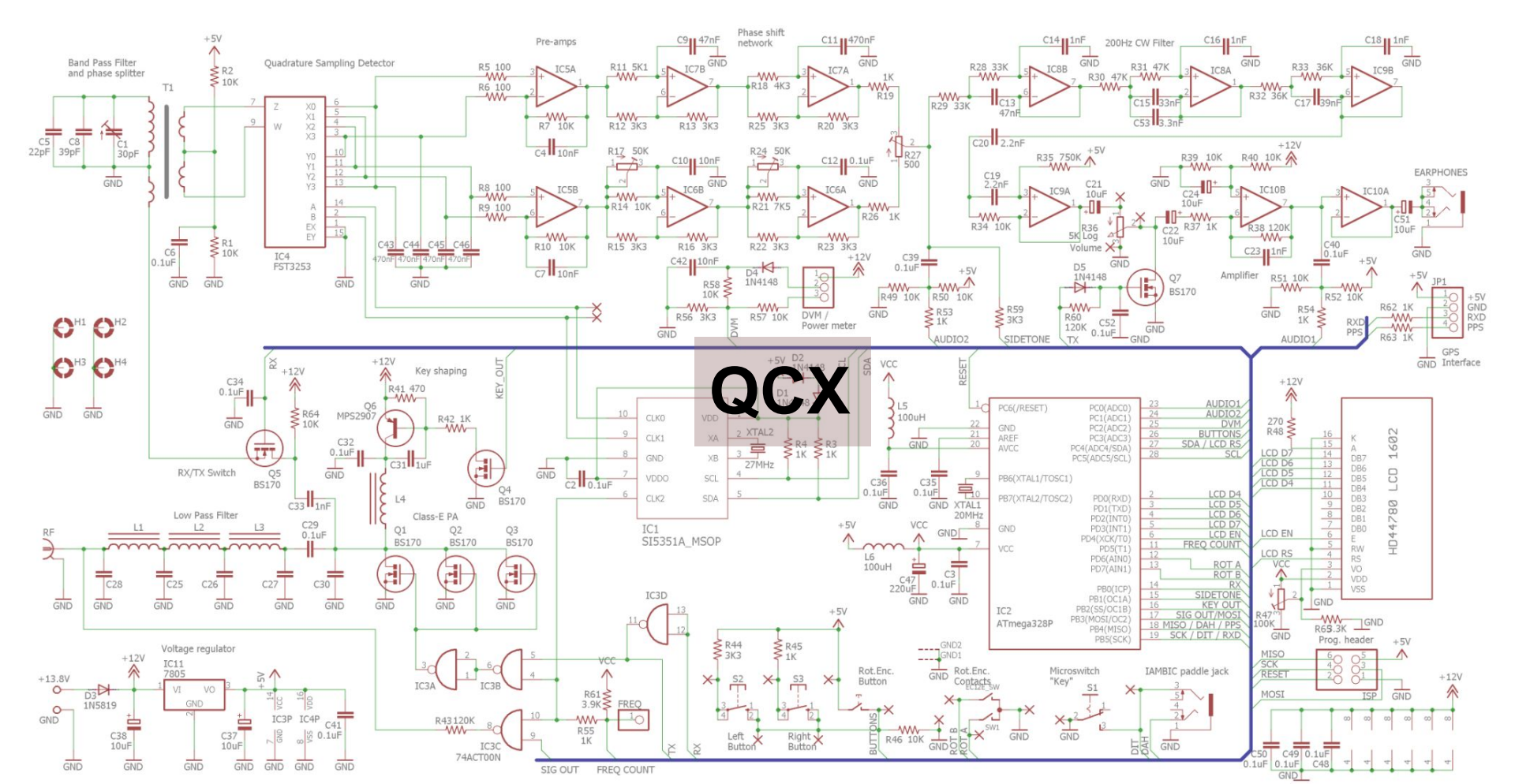
Analog Faz Cevrimi ile SSB yerine SDR
LSB/USB : Si5351 CLK0/CLK1 arasinda 90 derece faz kaydirmasi
ATMEGA328P over-sample ADC (62kHz) -> Hilbert-transform
10-bit ADC 72dB dinakim aralik li 2.4kHz SSB

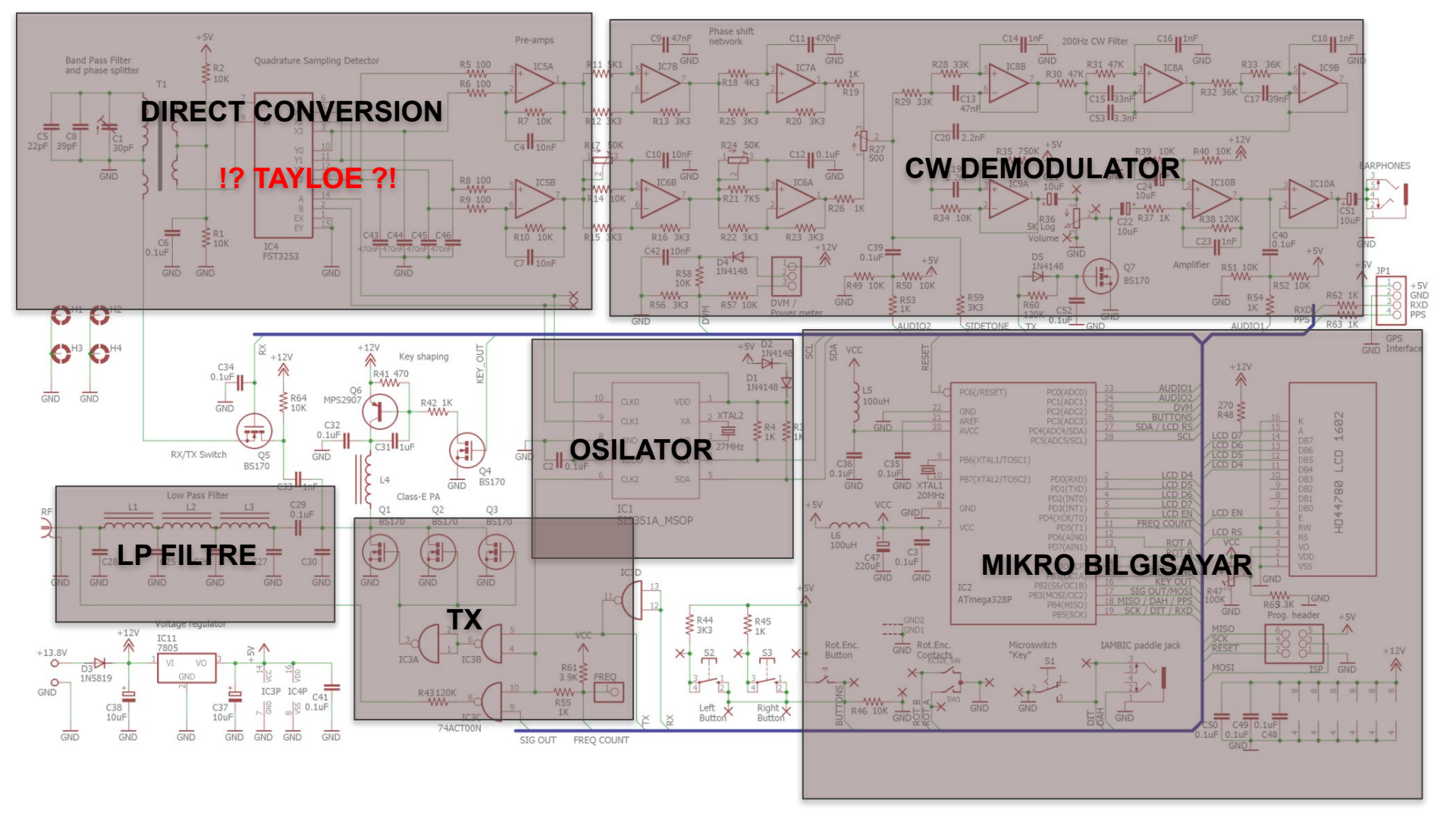




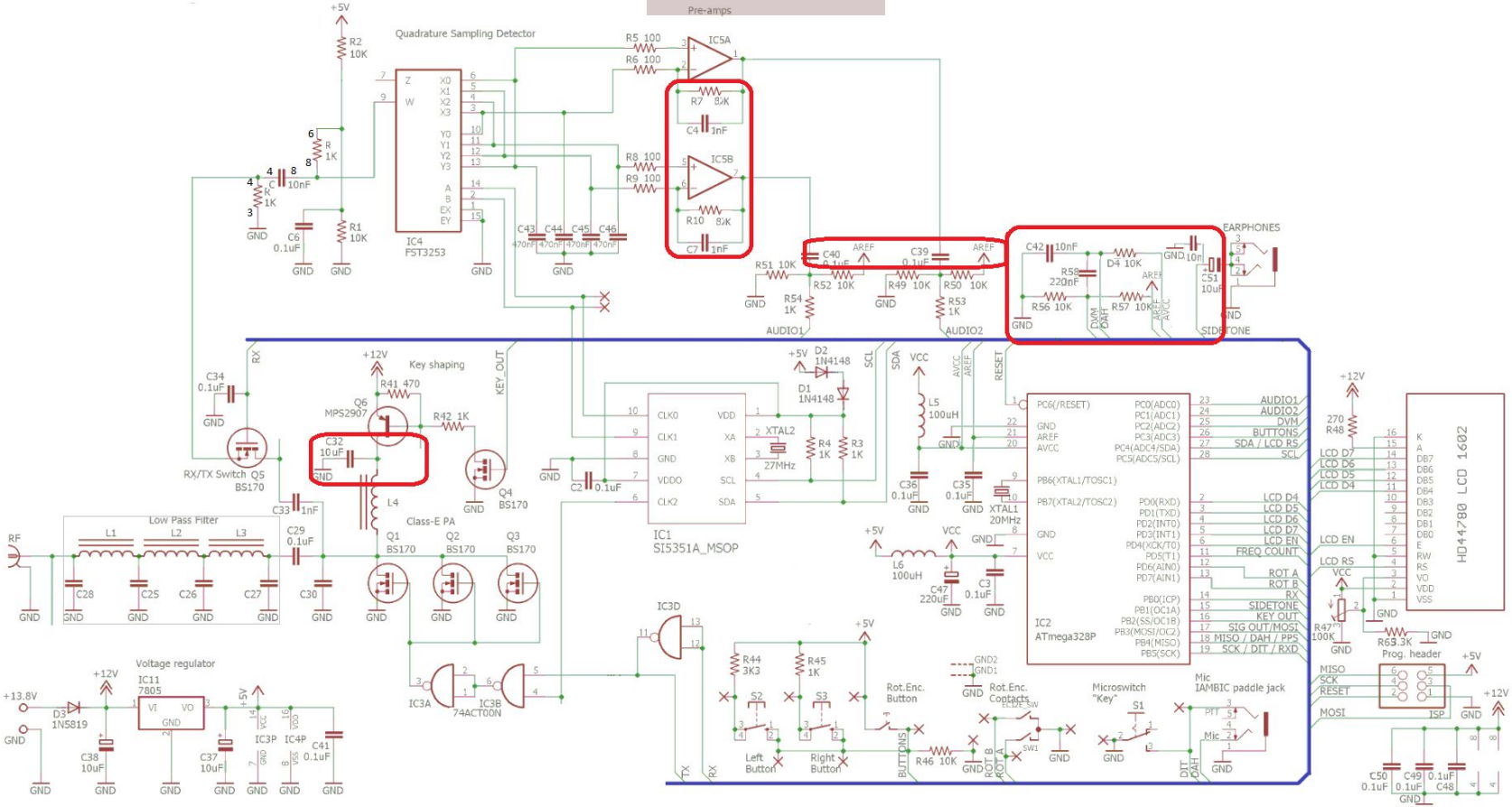
PE1NNZ - Guido

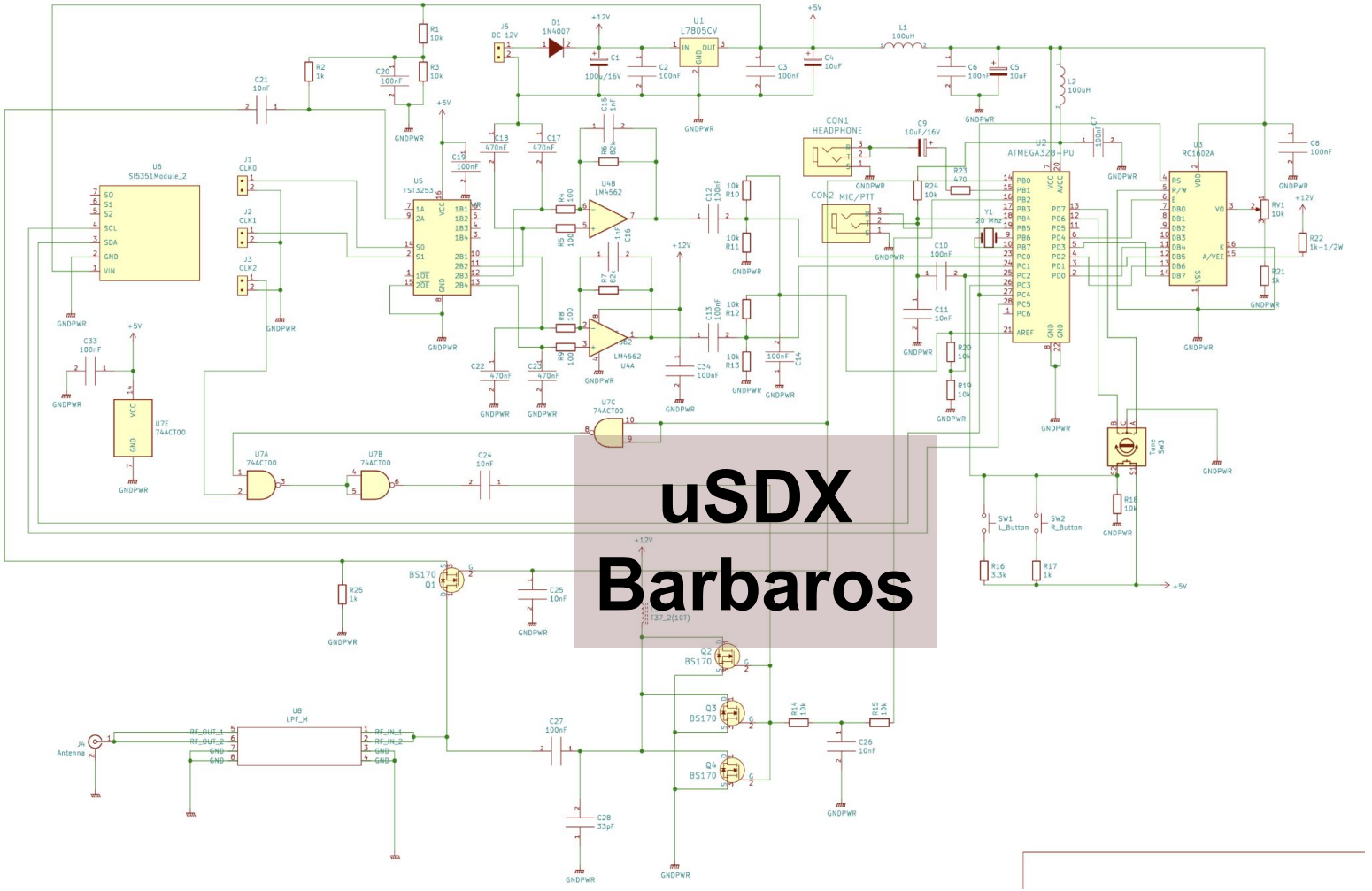


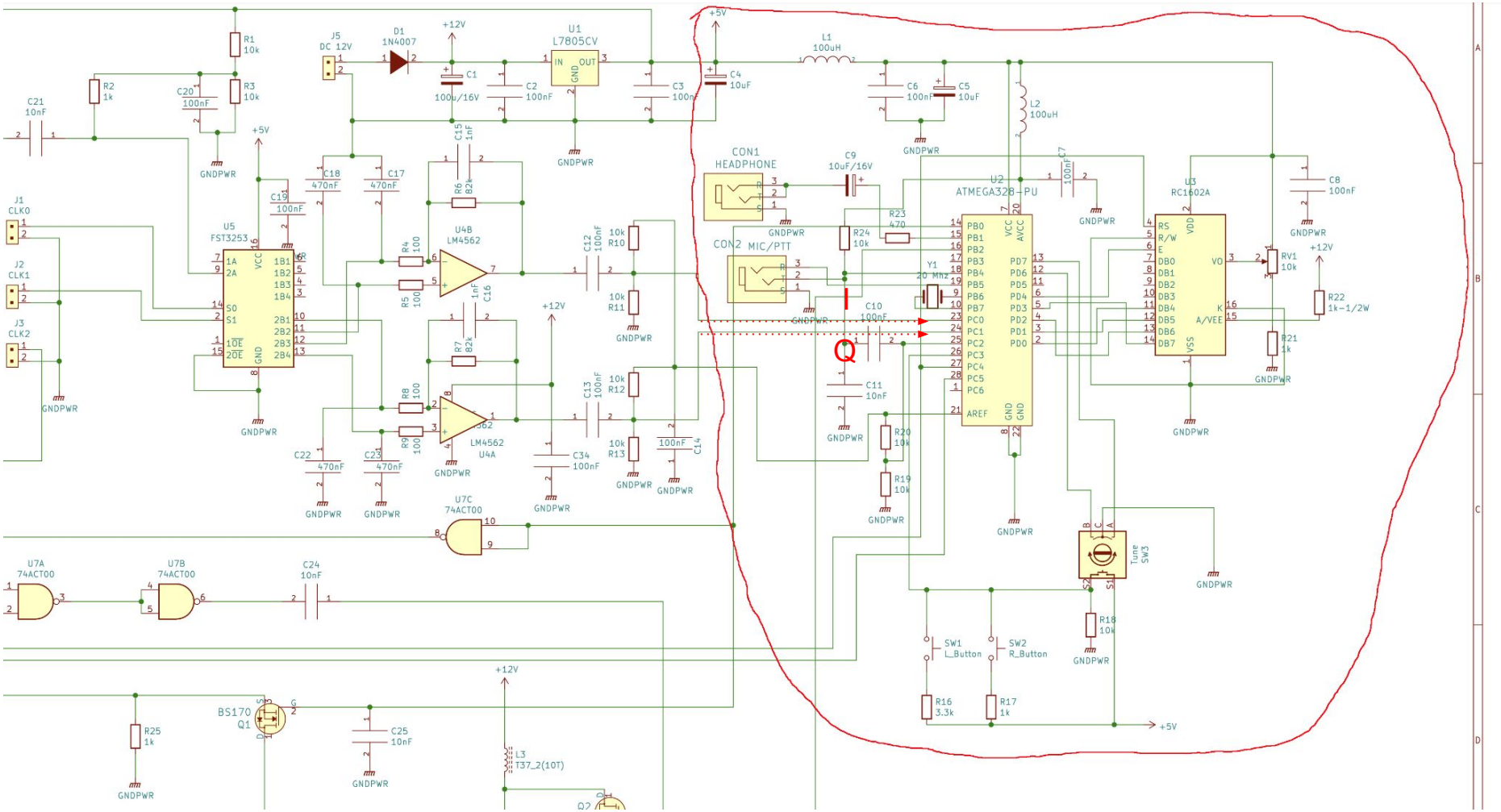

QCX
Band Pass Filter and phase splitter
Quadrature Sampling Detector
Pre-amps
Phase shift network
200Hz CW Filter
Amplifier
EARPHONES
Volume
Power meter
DVM
AUDIO1
AUDIO2
SIDETONE
GPS Interface
RX/TX Switch
Key shaping
Class-E PA
Low Pass Filter
Voltage regulator
Left Button
Right Button
Rot.Enc. Button
Rot.Enc. Contacts
Microswitch "Key"
IAMBIC paddle jack
Prog. header
SIG OUT
FREQ COUNT
IC1 SI5351A_MSOP
IC2 ATmega328P
HD44780 LCD 1602

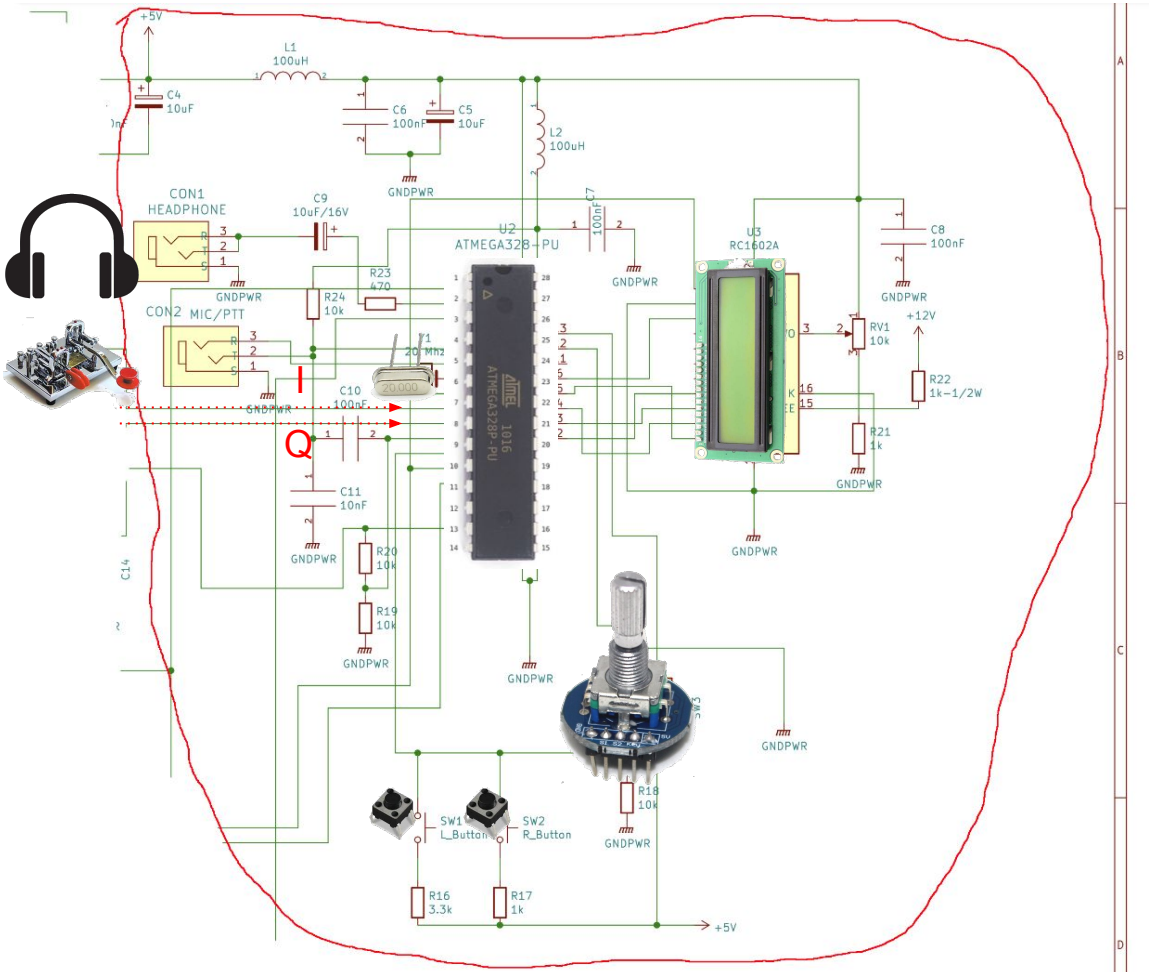

DIRECT CONVERSION
!? TAYLOE ?!
CW DEMODULATOR
OSILATOR
LP FILTRE
TX
MIKRO BILGISAYAR
Band Pass Filter and phase splitter
Quadrature Sampling Detector
Pre-amps
Phase shift network
200Hz CW Filter
Key shaping
Class-E PA
Low Pass Filter
Voltage regulator
RX/TX Switch
Amplifier
Volume
Left Button
Right Button
Rot.Enc. Button
Rot.Enc. Contacts
Microswitch "Key"
IAMBIC paddle jack
Prog. header
GPS Interface
ATmega328P
HD44780 LCD 1602
SIG OUT
FREQ COUNT

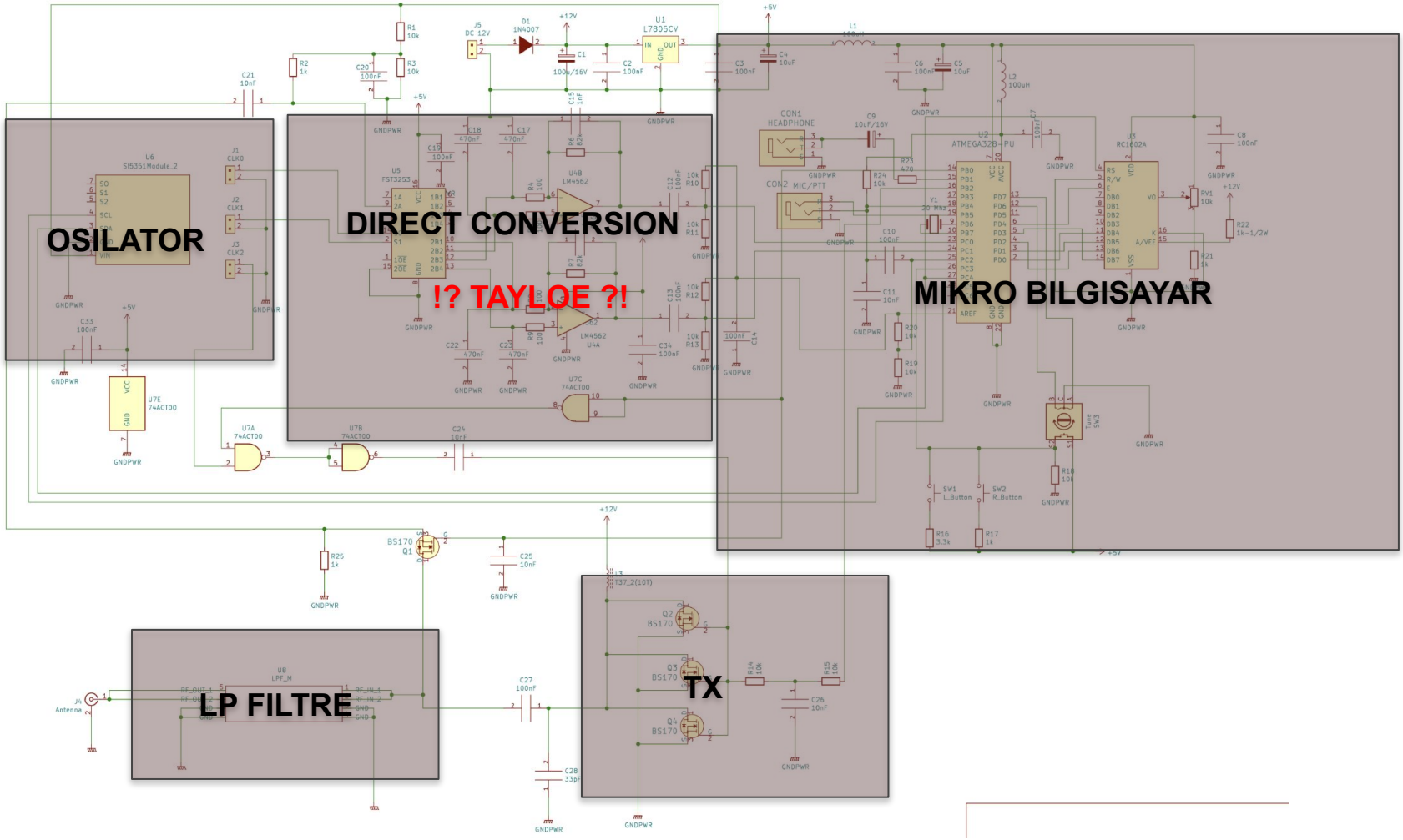
uSDX

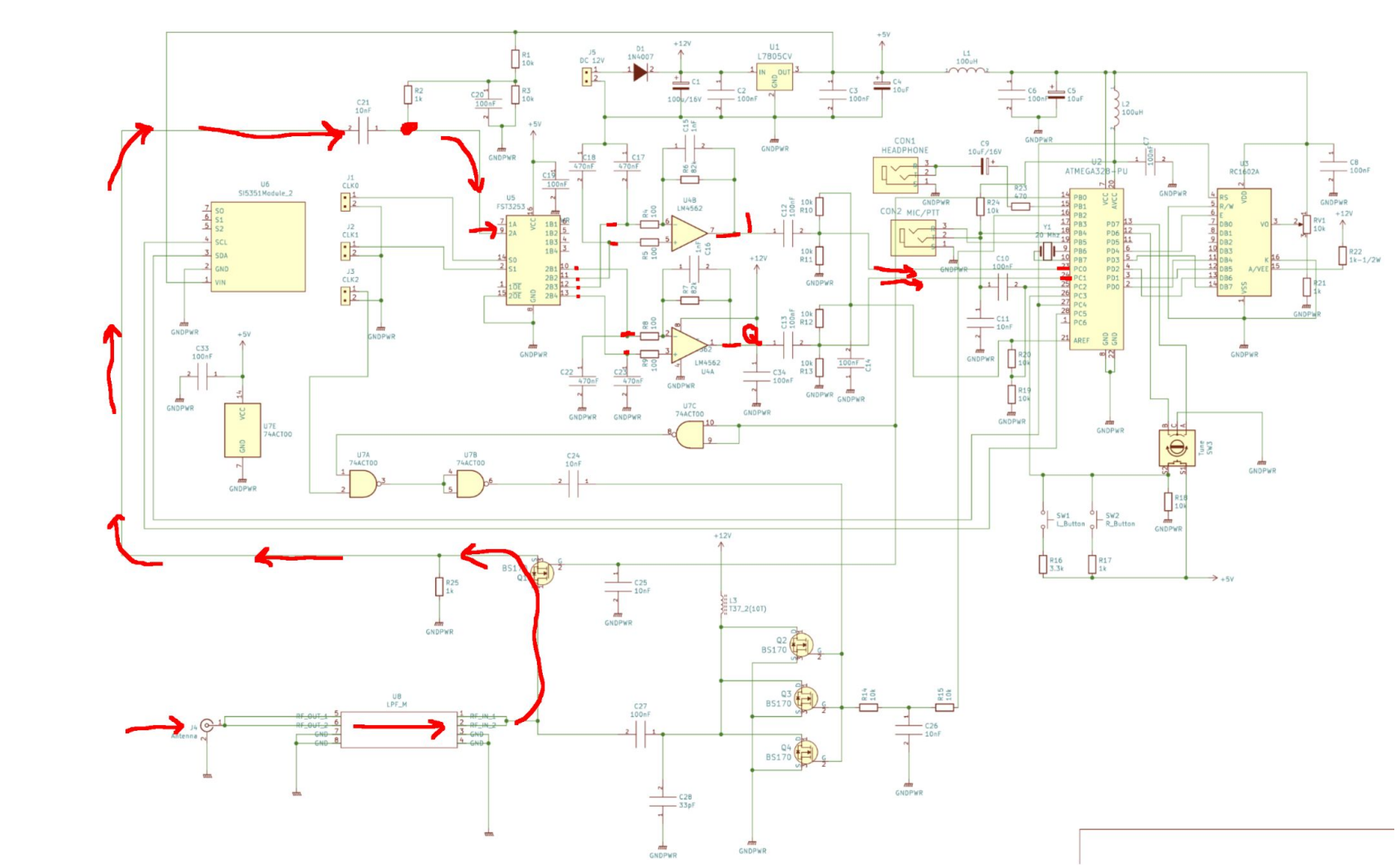

Pre-amps
Quadrature Sampling Detector
IC4 FST3253
IC5A
IC5B
EARPHONES
AUDIO1
AUDIO2
SIDETONE
Key shaping
RX/TX Switch Q5 BS170
Low Pass Filter
Class-E PA
Q1 BS170
Q2 BS170
Q3 BS170
Q4 BS170
Q6 MPS2907
IC1 SI5351A_MSOP
IC2 ATmega328P
IC3A IC3B 74ACT00N
IC3D
Voltage regulator
IC11 7805
Left Button
Right Button
Rot.Enc. Button
Rot.Enc. Contacts
Microswitch "Key"
Mic IAMBIC paddle jack
HD44780 LCD 1602
Prog. header
ISP

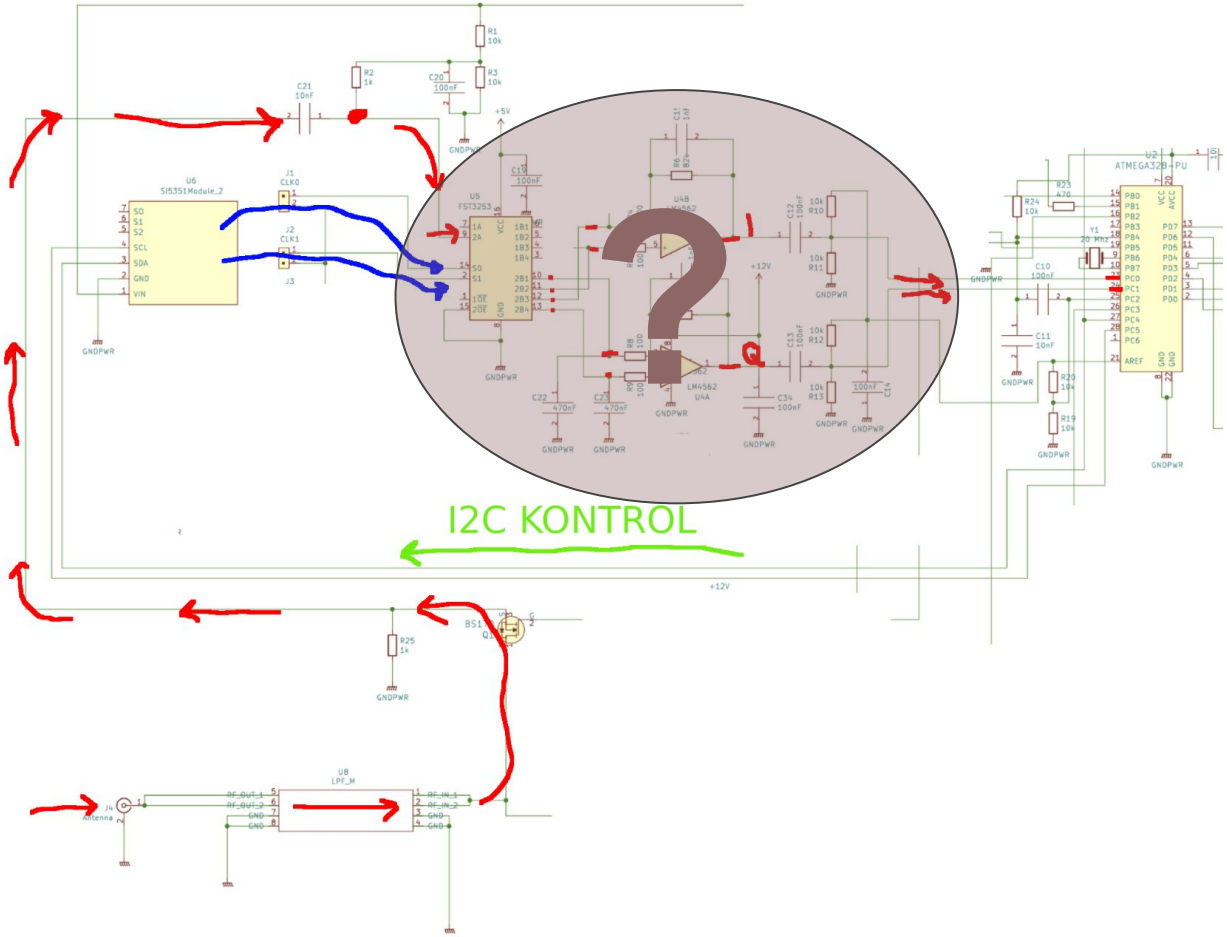

uSDX
Barbaros

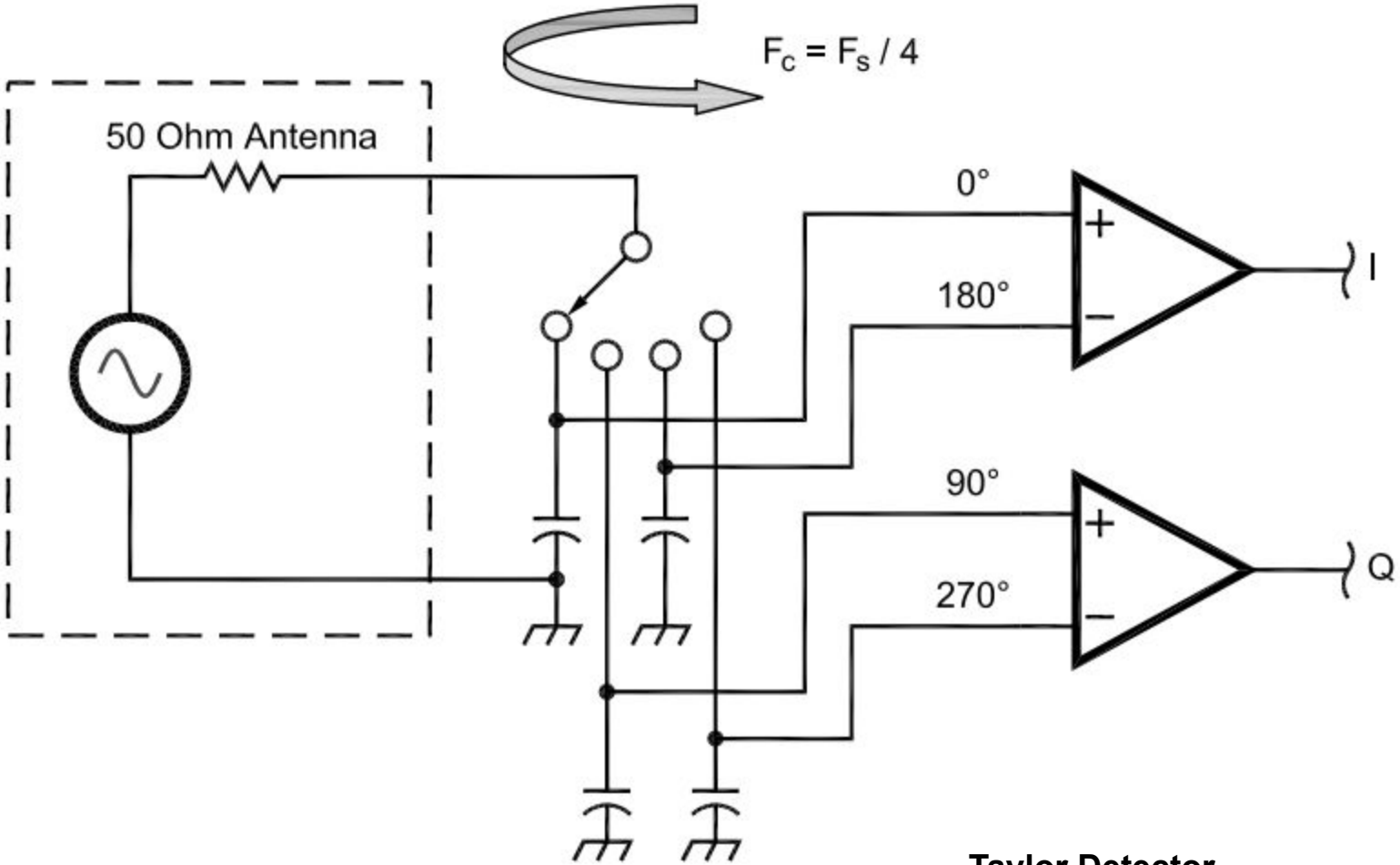

J5
DC 12V
D1
1N4007
+12V
U1
L7805CV
IN
OUT
GND
+5V
L1
100uH
L2
100uH
C1
100u/16V
C2
100nF
C3
100nF
C4
10uF
C5
10uF
C6
100nF
C7
100nF
C8
100nF
C9
10uF/16V
C10
100nF
C11
10nF
C12
100nF
C13
100nF
C14
100nF
C15
1nF
C16
1nF
C17
470nF
C18
470nF
C19
100nF
C20
100nF
C21
10nF
C22
470nF
C23
470nF
C24
10nF
C25
10nF
C34
100nF
R1
10k
R2
1k
R3
10k
R4
100
R5
100
R6
82k
R7
82k
R8
100
R9
100
R10
10k
R11
10k
R12
10k
R13
10k
R16
3.3k
R17
1k
R18
10k
R19
10k
R20
10k
R21
1k
R22
1k-1/2W
R23
470
R24
10k
R25
1k
RV1
10k
GNDPWR
J1
CLK0
J2
CLK1
J3
CLK2
U5
FST3253
VCC
GND
1A
2A
1B1
1B2
1B3
1B4
2B1
2B2
2B3
2B4
S0
S1
1OE
2OE
U4B
LM4562
U4A
LM4562
U7A
74ACT00
U7B
74ACT00
U7C
74ACT00
CON1
HEADPHONE
CON2
MIC/PTT
U2
ATMEGA328-PU
PB0
PB1
PB2
PB3
PB4
PB5
PB6
PB7
PC0
PC1
PC2
PC3
PC4
PC5
PC6
AREF
PD7
PD6
PD5
PD4
PD3
PD2
PD1
PD0
Y1
20 MHz
U3
RC1602A
RS
R/W
E
DB0
DB1
DB2
DB3
DB4
DB5
DB6
DB7
VDD
VO
K
A/VEE
VSS
SW1
L_Button
SW2
R_Button
Tune
SW3
Q1
BS170
L3
T37_2(10T)
Q2
Q


+5V
L1
100uH
C4
10uF
C6
100nF
C5
10uF
L2
100uH
GNDPWR
CON1
HEADPHONE
C9
10uF/16V
U2
ATMEGA328-PU
C7
100nF
U3
RC1602A
C8
100nF
R23
470
R24
10k
CON2 MIC/PTT
Y1
20 Mhz
20.000
C10
100nF
RV1
10k
+12V
R22
1k-1/2W
R21
1k
C11
10nF
R20
10k
R19
10k
SW1
L_Button
SW2
R_Button
R18
10k
R16
3.3k
R17
1k
I
Q


OSILATOR
DIRECT CONVERSION
!? TAYLOE ?!
MIKRO BILGISAYAR
LP FILTRE
TX


J5
DC 12V
D1
1N4007
U1
L7805CV
L1
100uH
L2
100uH
CON1
HEADPHONE
CON2
MIC/PTT
U2
ATMEGA328-PU
U3
RC1602A
U4
LM4562
U5
FST3253
U6
SI5351Module_2
U7
74ACT00
U8
LPF_M
Y1
20 Mhz
SW1
L_Button
SW2
R_Button
Tune
SW3
J4
Antenna
L3
T37_2(10T)
Q1
BS170
Q2
BS170
Q3
BS170
Q4
BS170
GNDPWR
+5V
+12V


I2C KONTROL
U6
SI5351Module_2
SO
S1
S2
SCL
SDA
GND
VIN
U5
FST3253
U4B
LM4562
U4A
U2
ATMEGA328-PU
J1
CLK0
J2
CLK1
J3
J4
Antenna
U8
LPF_M
BS1
Q3
Y1
20 MHz
+5V
+12V
GNDPWR
R1
10k
R2
1k
R3
10k
R23
470
R24
10k
R25
1k
C20
100nF
C21
10nF
C10
100nF
C11
10nF
C12
100nF
C13
100nF
C14
100nF
C22
470nF
C23
470nF
C34
100nF
R10
10k
R11
10k
R12
10k
R13
10k
R19
10k
R20
10k



Taylor Detector

Tayloe Quadrature Product Detector

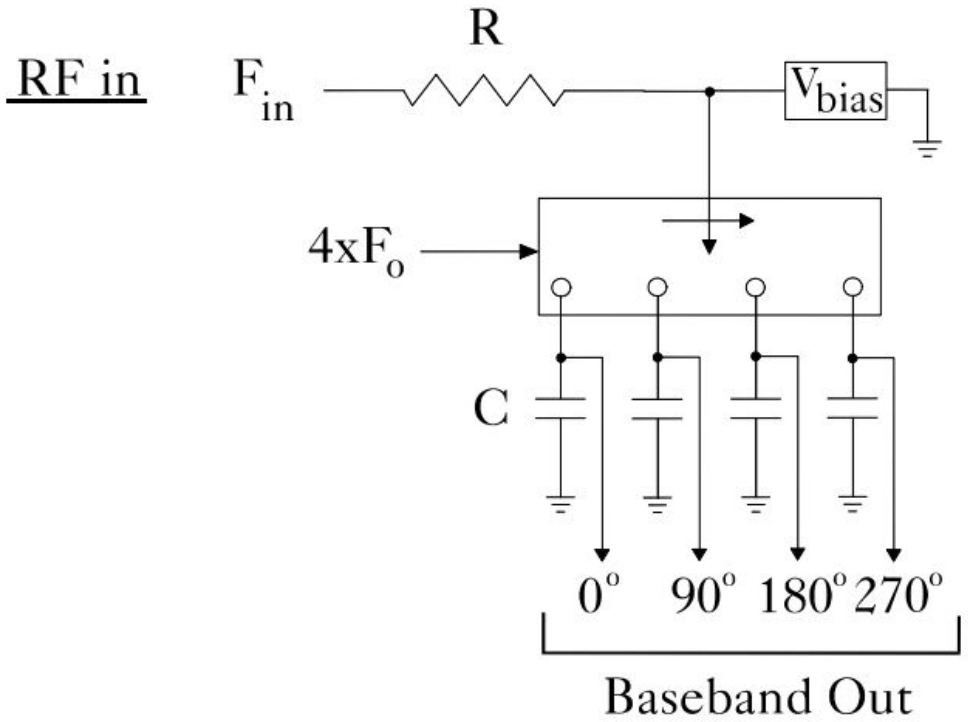
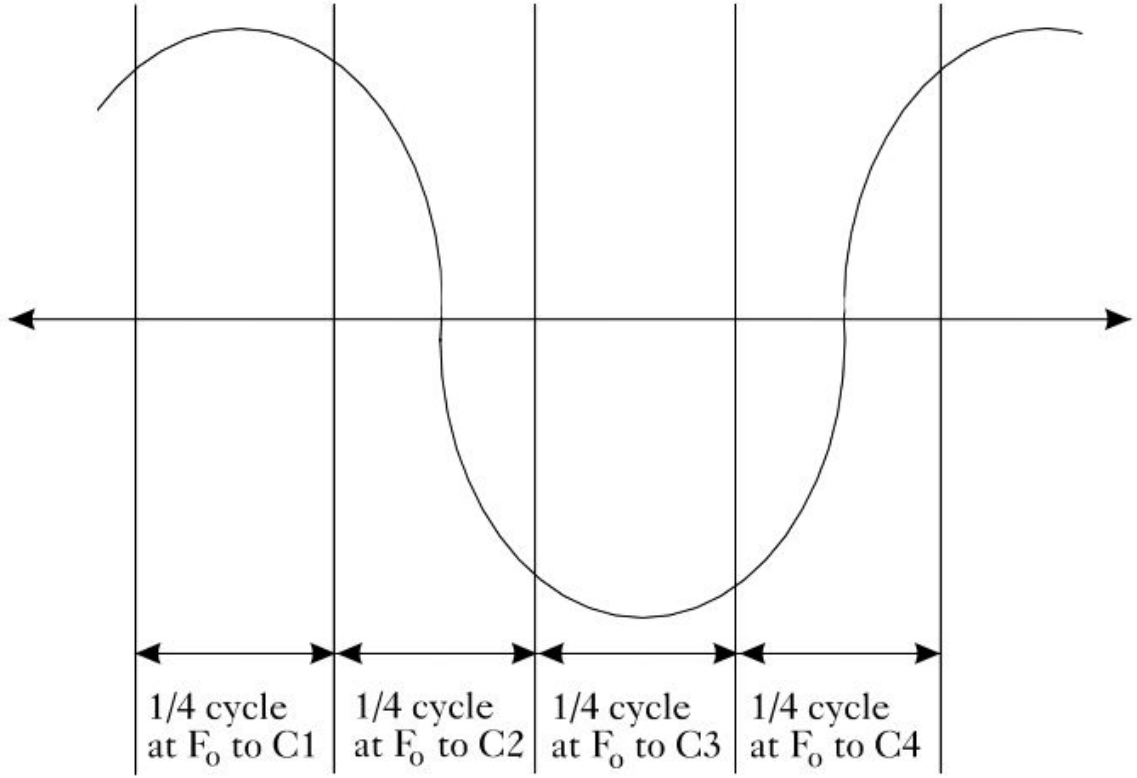
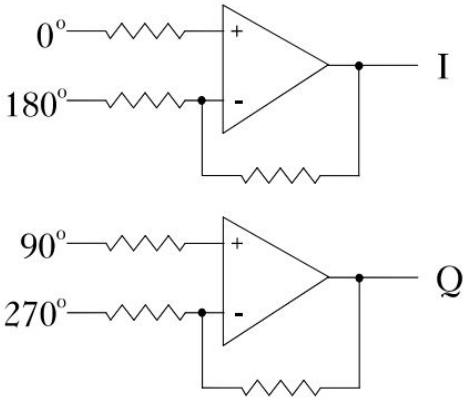


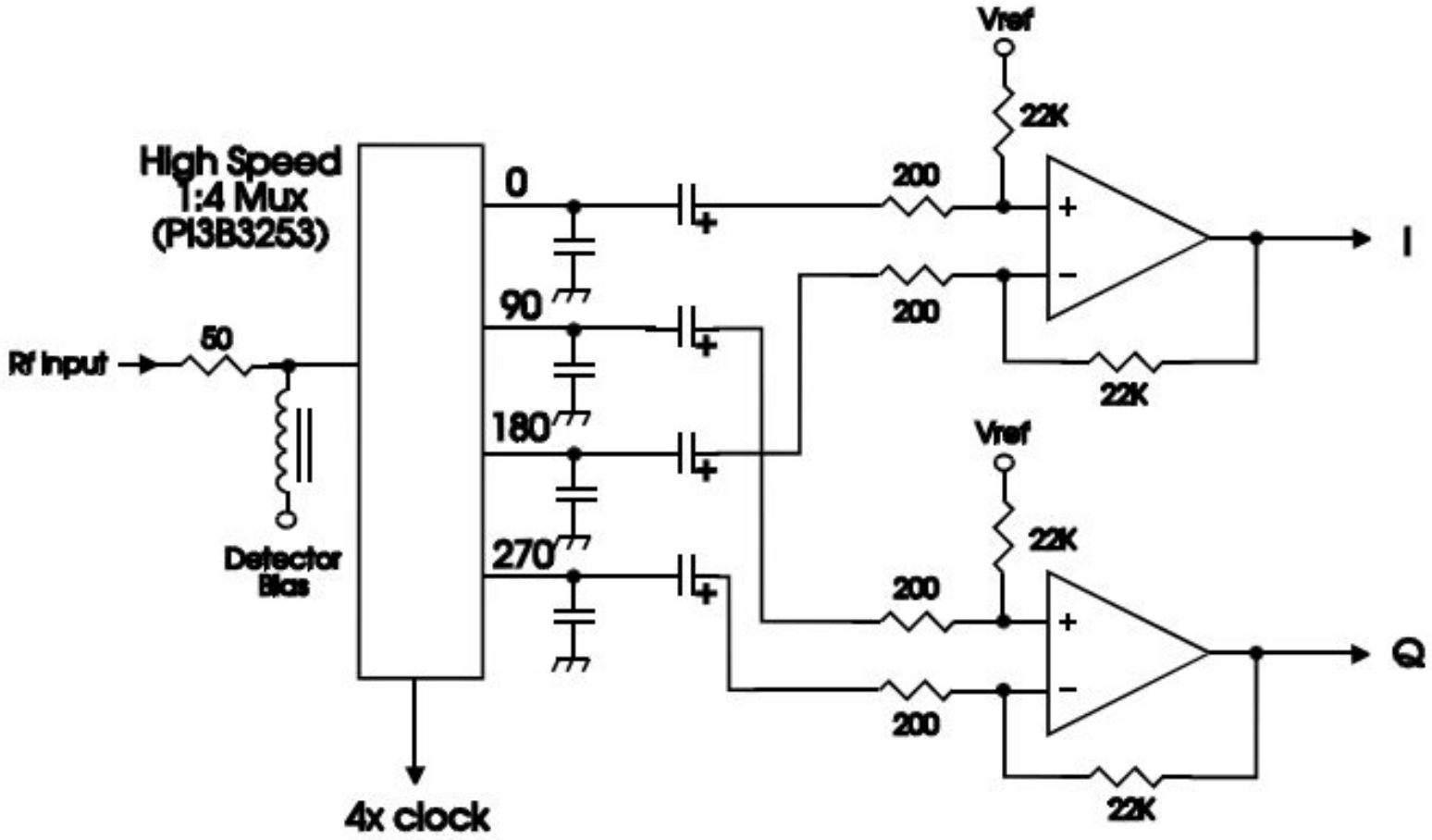
Figure # 1 Tayloe Quadrature Product Detector

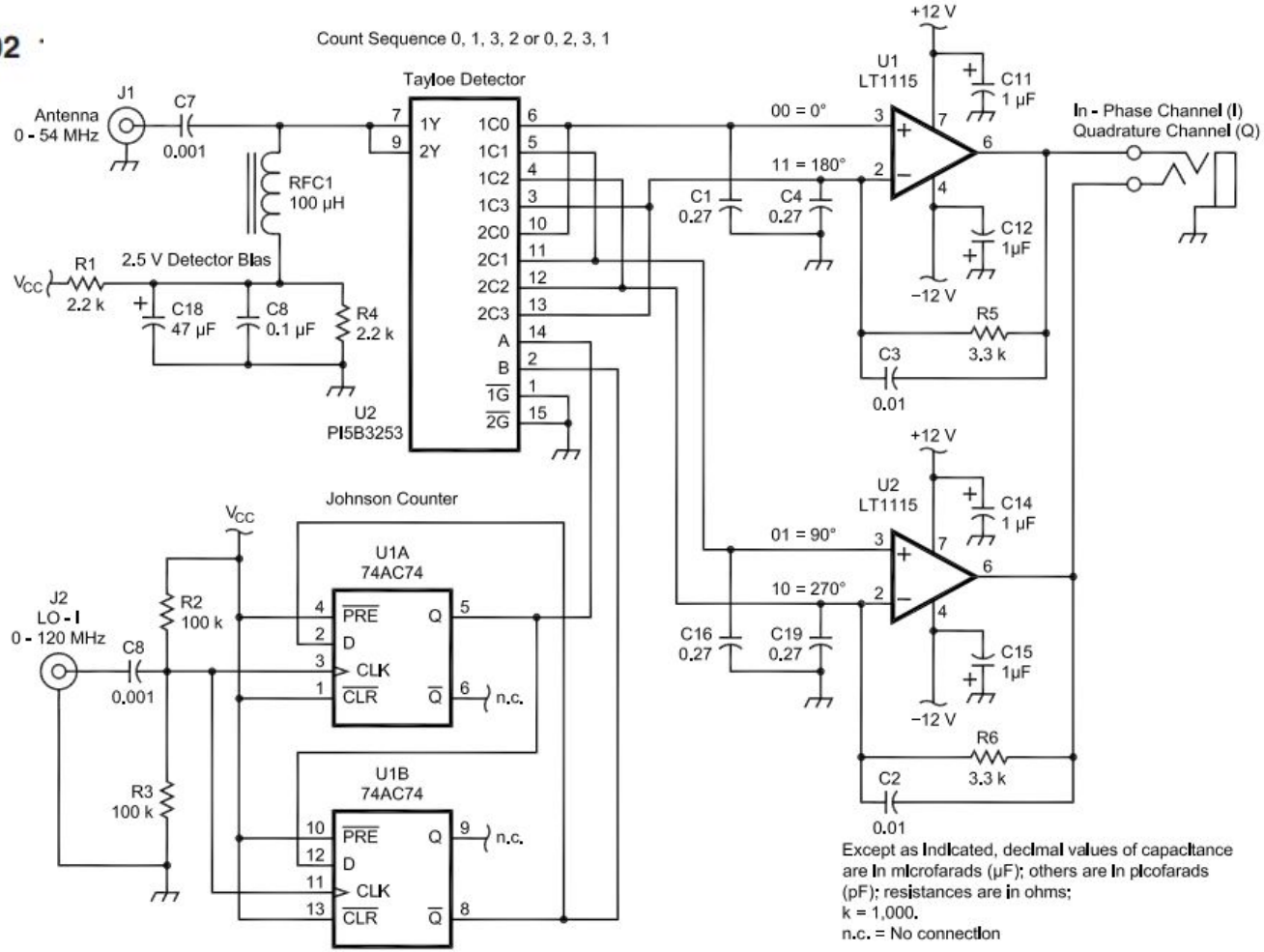
http://www.norcalqrp.org/files/Tayloe_mixer_x3a.pdf



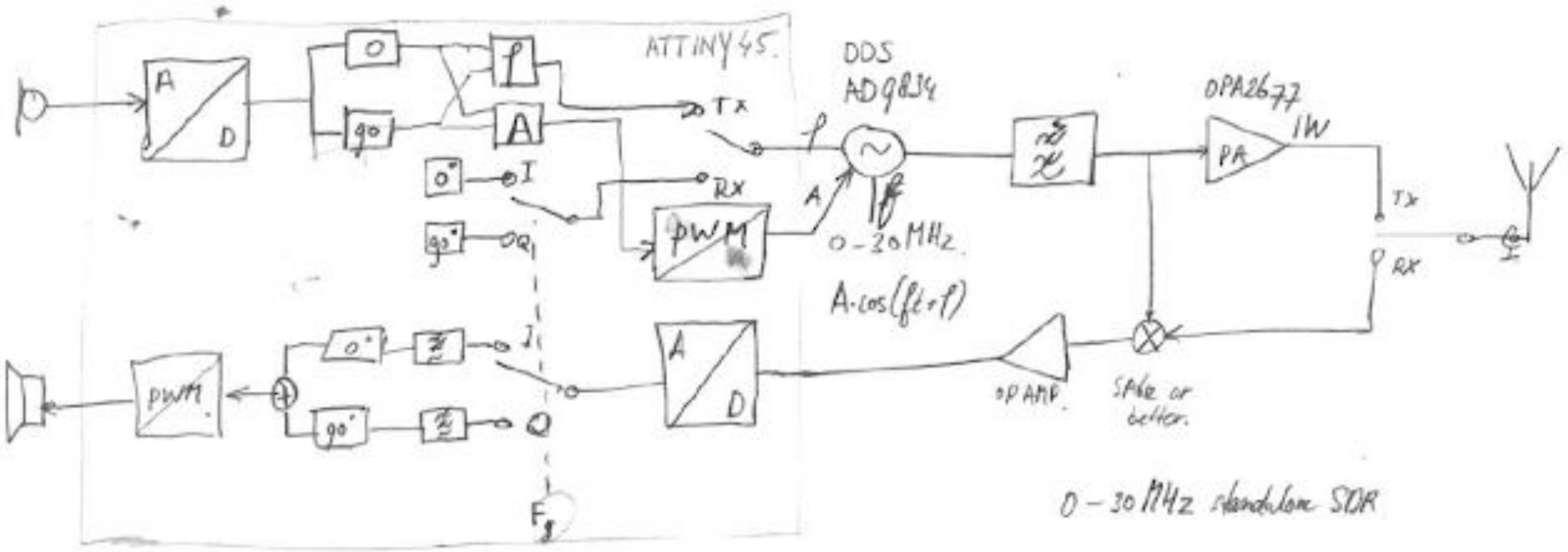


igure # 2 One cycle Sine Wave at Sampled Frequency F_o


High Speed
1:4 Mux
(PI3B3253)
Rf Input
50
Detector
Bias
0
90
180
270
4x clock
Vref
22K
200
200
22K
I
Vref
22K
200
200
22K
Q

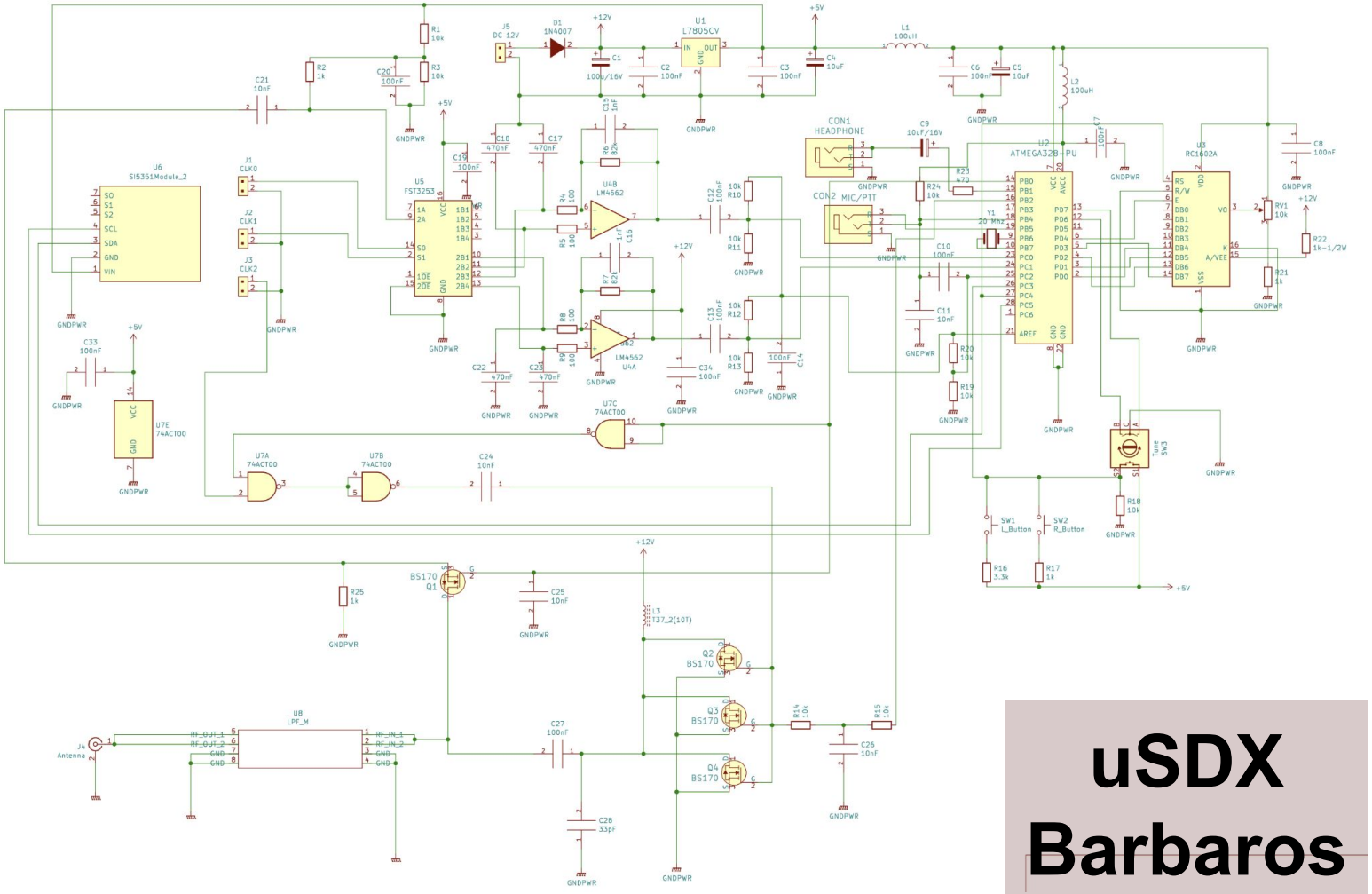


Except as indicated, decimal values of capacitance are in microfarads (µF); others are in picofarads (pF); resistances are in ohms;
k = 1,000.
n.c. = No connection

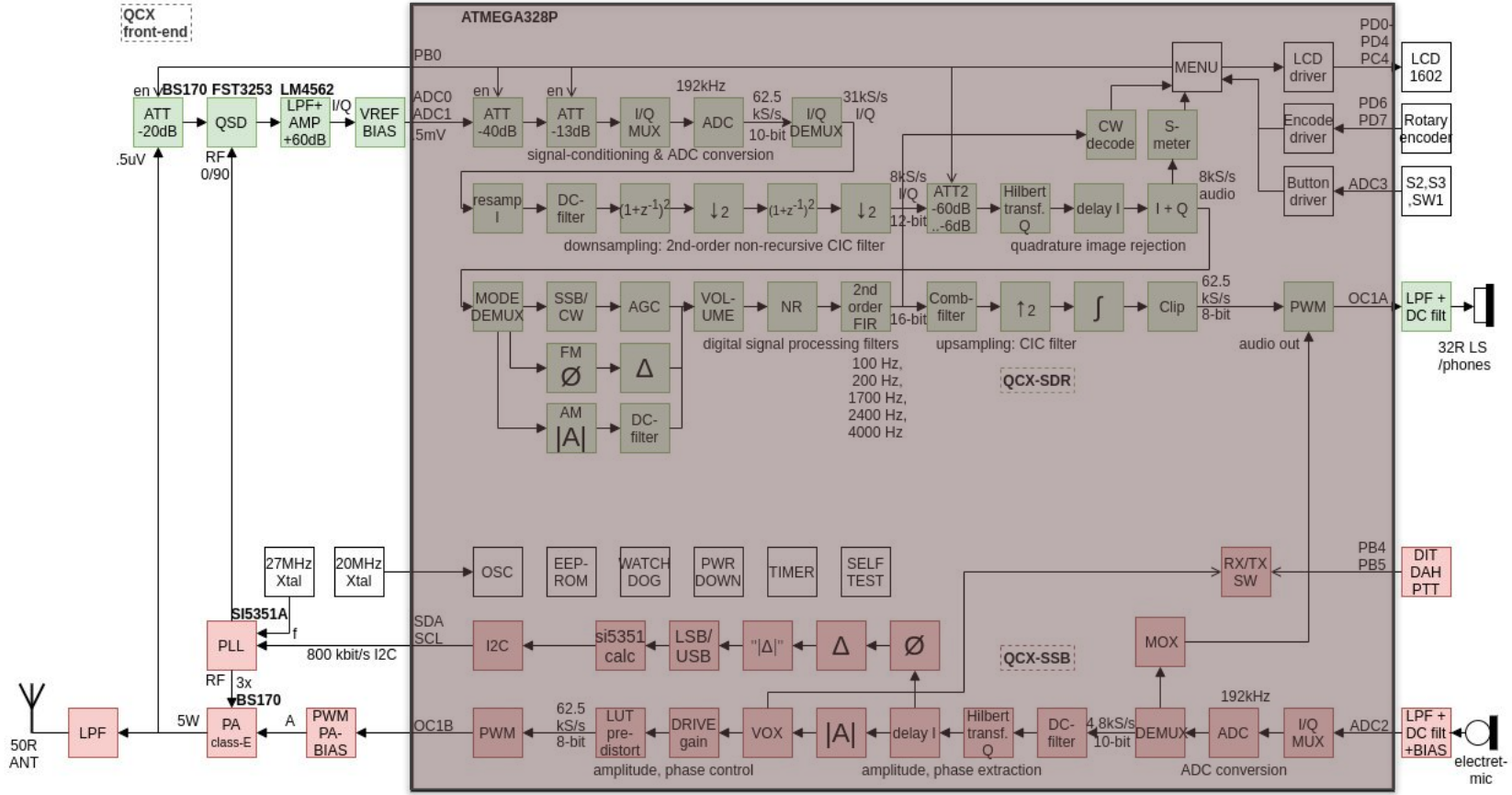


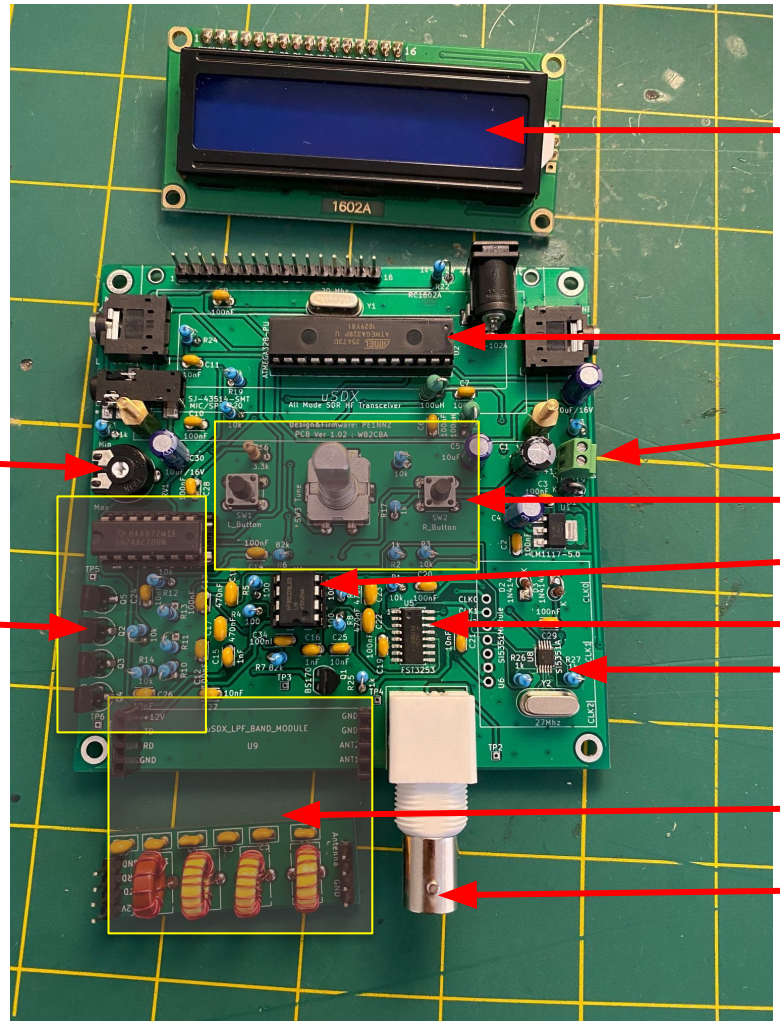
I'm Guido, radio amateur licensed as PE1NNZ, living in south of Netherlands, i like qrp, homebrewing, digital communications, being reachable by email at pe1nnz@amsat.org

PE1NNZ Direct SSB generation by frequency modulating a PLL


uSDX
Barbaros

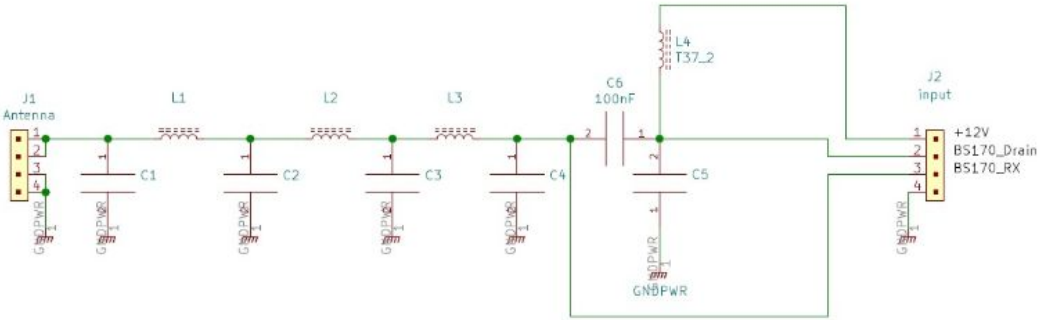
Ve ARTIK HERSEY YAZILIM

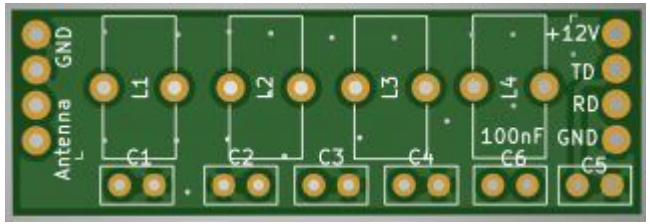


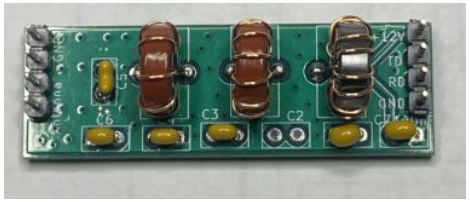

EKRAN
ISLEMCI
12V BESLEME
KONTROLLER
OPAMP
FST3253 ANAHTARLAYICI
Si5351 OSILATOR
ALCAK GECIREN FILTRE
ANTEN GIRISI
EKRAN PARLAKLIK
VERICI
1602A
uSDX
SDX_LPF_BAND_MODULE

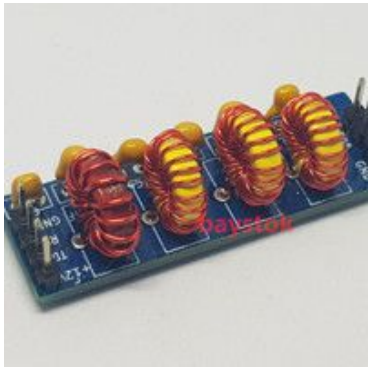
Filtreler

CIN MALI TOROID KULLANMAMANIZI ONERIRIM !!!









80m Band:

L1-T37-2 / 25 turns
L2-T37-2 / 27 turns
L3- T37-2/27 turns
L4- T37-2/24 turns
C1-470pf
C2-1200pf
C3- 1200pf
C4-470pf
C5-180pf

40m Band:

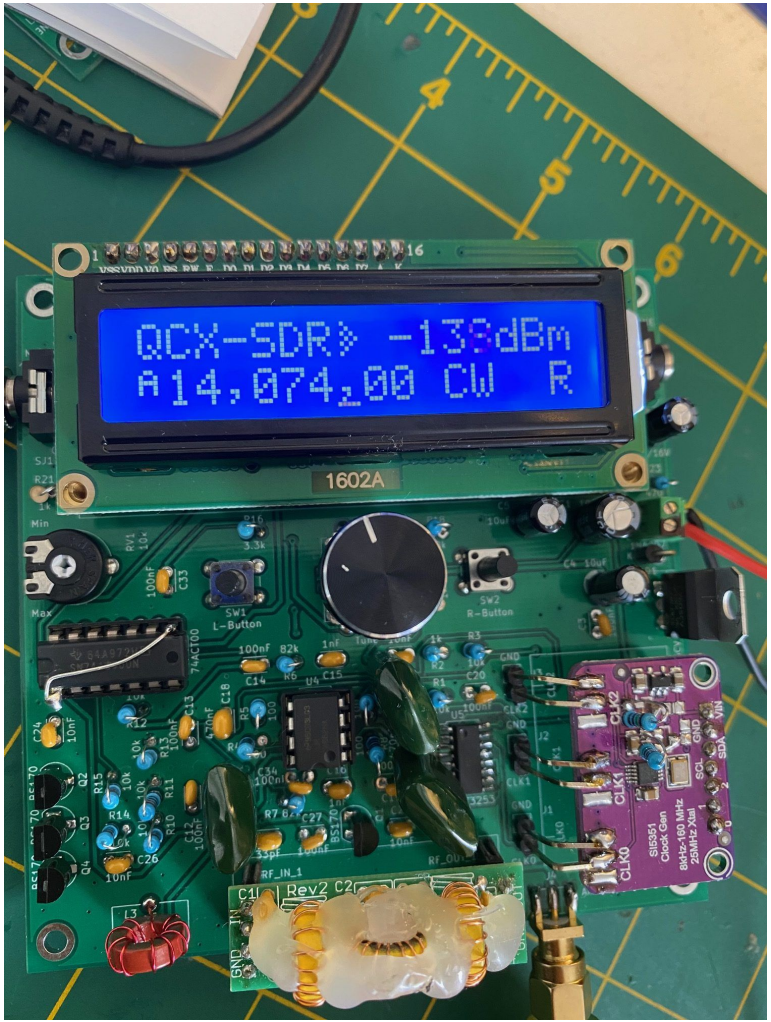
L1-T37-6 / 21 turns
L2-T37-6 / 24 turns
L3- T37-6/21 turns
L4- T37-2/16 turns
C1-270pf
C2-680pf
C3- 680pf
C4-270pf
C5-56pf

30m Band:

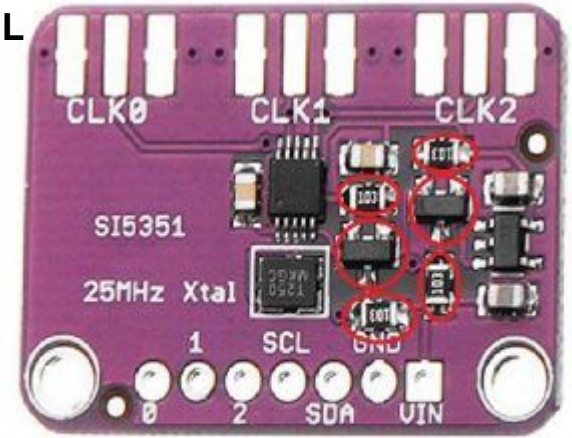
L1-T37-6 / 19 turns
L2-T37-6 / 20 turns
L3- T37-6/19 turns
L4- T37-2/14 turns
C1-270pf
C2-560pf
C3- 560pf
C4-270pf
C5-30pf

20m Band:

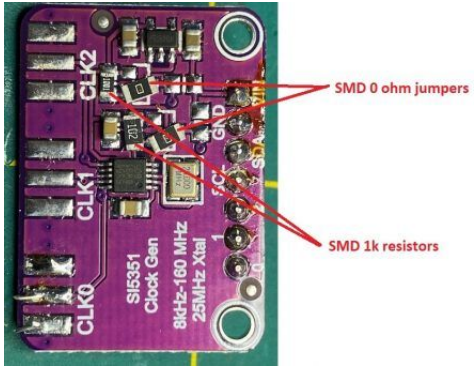
L1-T37-6 / 16 turns
L2-T37-6 / 17 turns
L3- T37-6/16 turns
L4- T37-2/10 turns
C1-180pf
C2-390pf
C3- 390pf
C4-180pf
C5-30pf

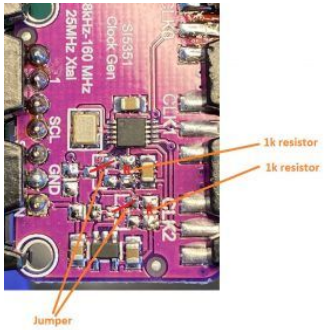


SI5351 MODUL KULLANIMI



Remove red circled components



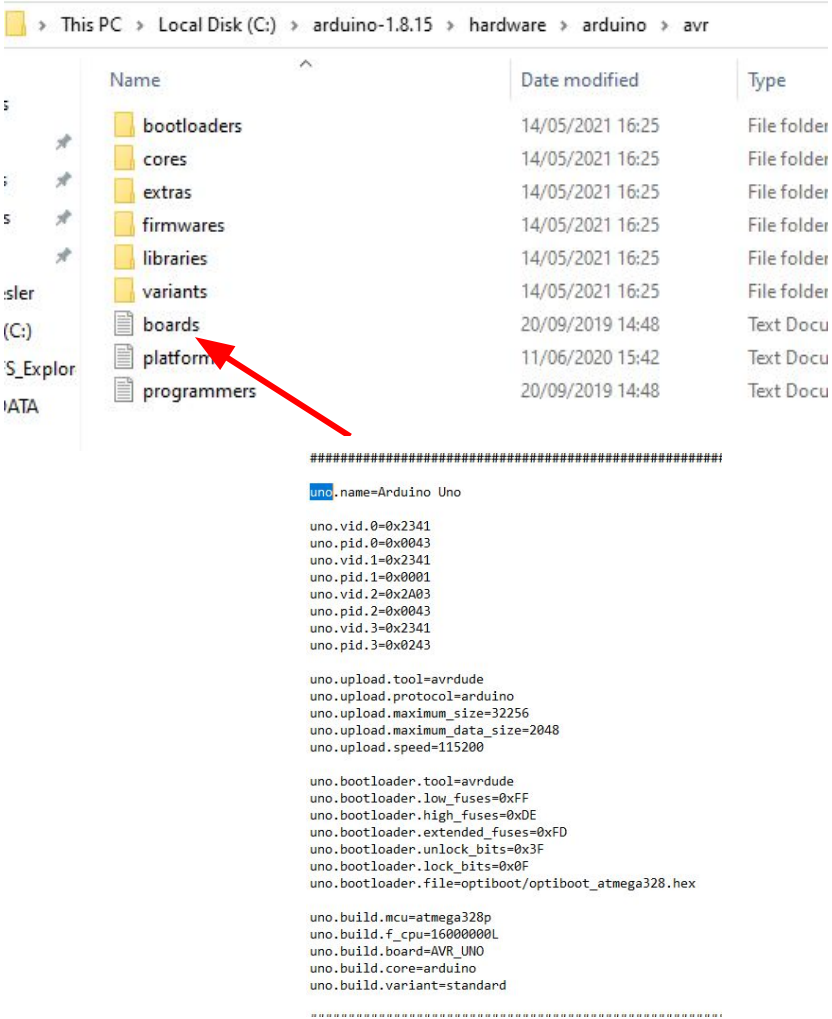


Programlama

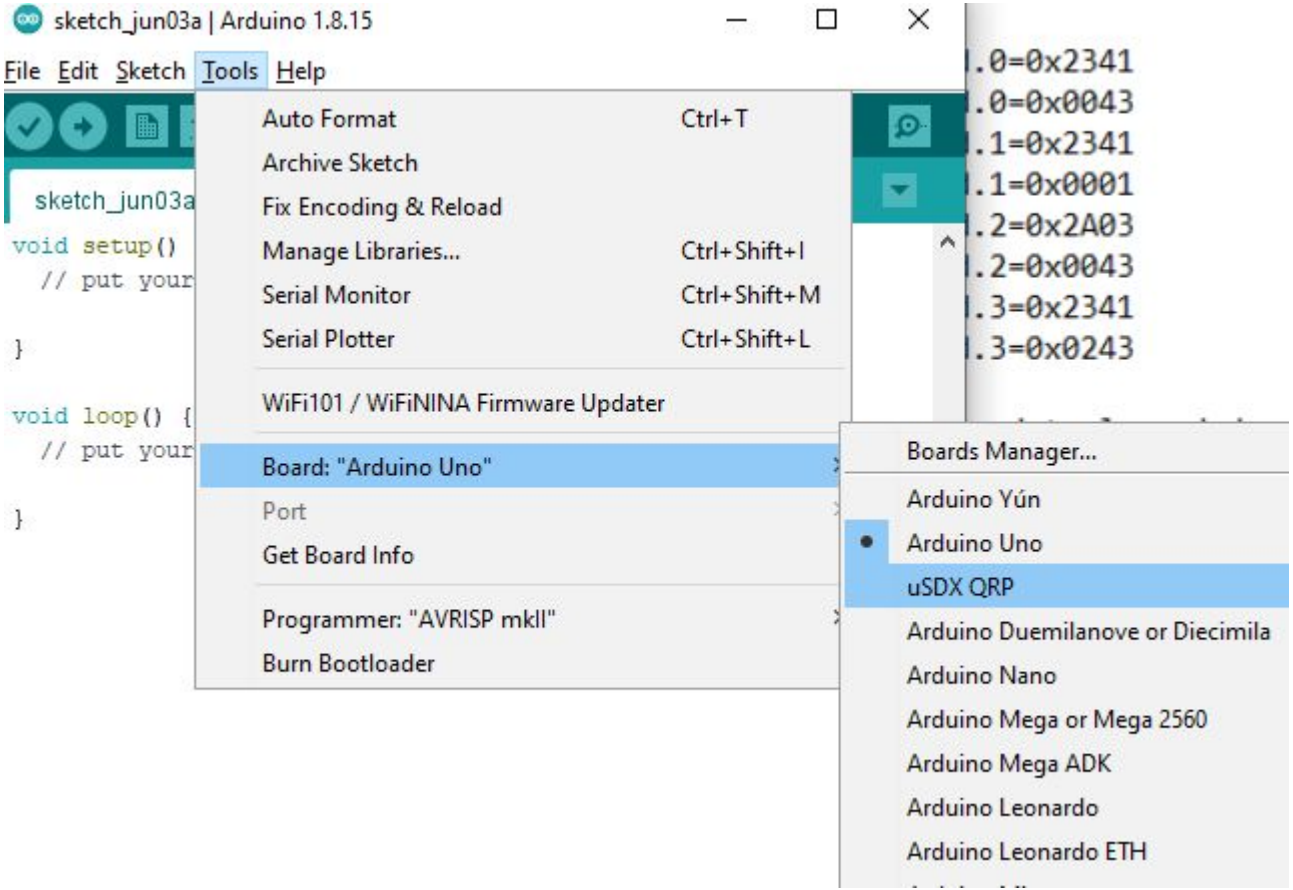
```
avrdude -c avrisp -b 19200 -P /dev/ttyUSB0 -p m328p -e
-U efuse:w:0xFD:m -U hfuse:w:0xD6:m -U lfuse:w:0xFF:m -U flash:w:R1.0x.hex
```

FUSE

```
E=FD
H=D6
L=FF
```

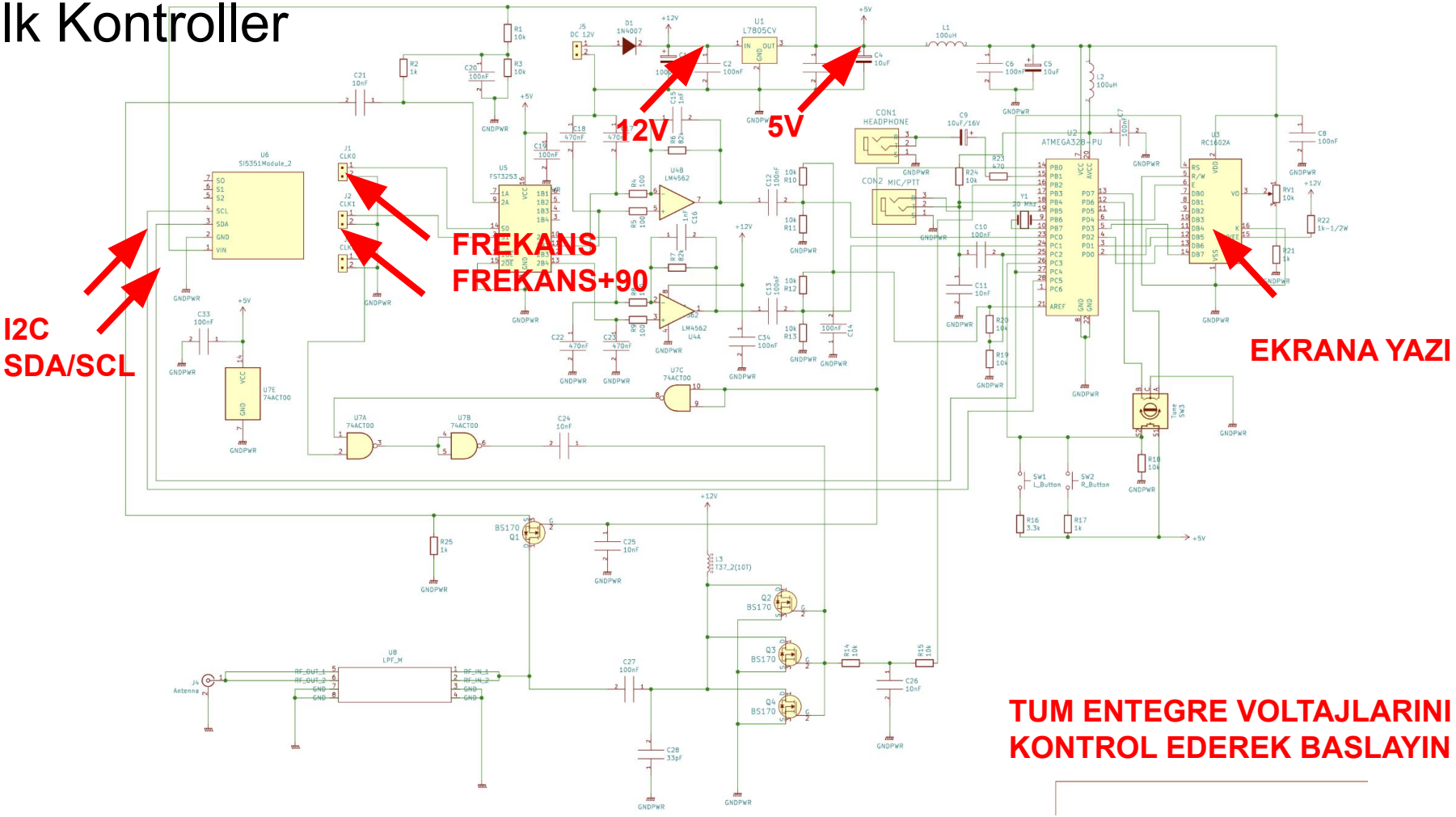


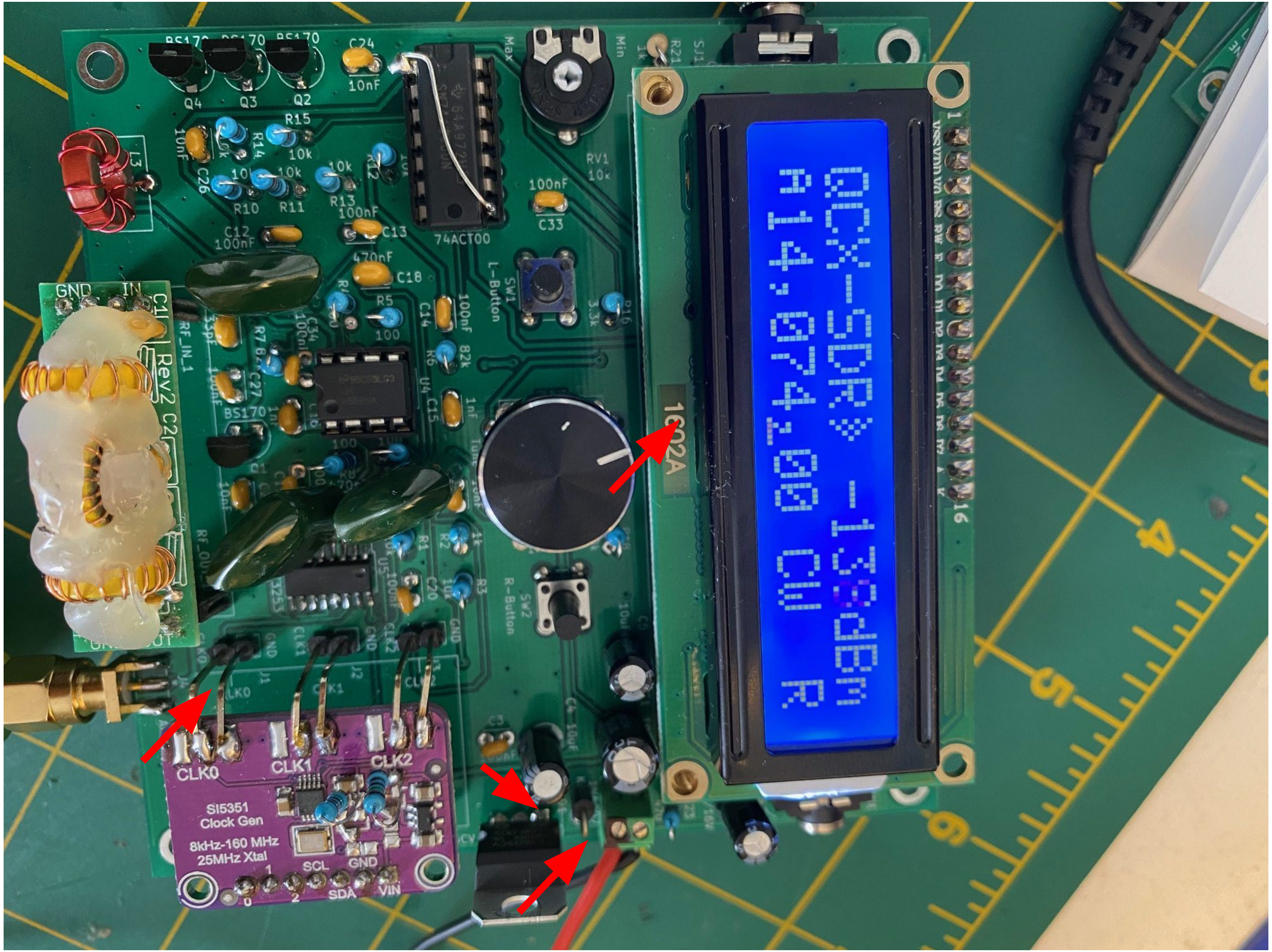

```
usdx.name=uSDX QRP

usdx.vid.0=0x2341
usdx.pid.0=0x0043
usdx.vid.1=0x2341
usdx.pid.1=0x0001
usdx.vid.2=0x2A03
usdx.pid.2=0x0043
usdx.vid.3=0x2341
usdx.pid.3=0x0243

usdx.upload.tool=avrdude
usdx.upload.protocol=arduino
usdx.upload.maximum_size=32256
usdx.upload.maximum_data_size=2048
usdx.upload.speed=115200

usdx.bootloader.tool=avrdude
usdx.bootloader.low_fuses=0xFF
usdx.bootloader.high_fuses=0xD6
usdx.bootloader.extended_fuses=0xFD
usdx.bootloader.unlock_bits=0x3F
usdx.bootloader.lock_bits=0x0F
usdx.bootloader.file=optiboot/optiboot_atmega328.h

usdx.build.mcu=atmega328p
usdx.build.f_cpu=16000000L
usdx.build.board=AVR_UNO
usdx.build.core=arduino
usdx.build.variant=standard
```



20 MHz bootloader yuklenmisse direk yazdirabilirsiniz

Ilk Kontroller



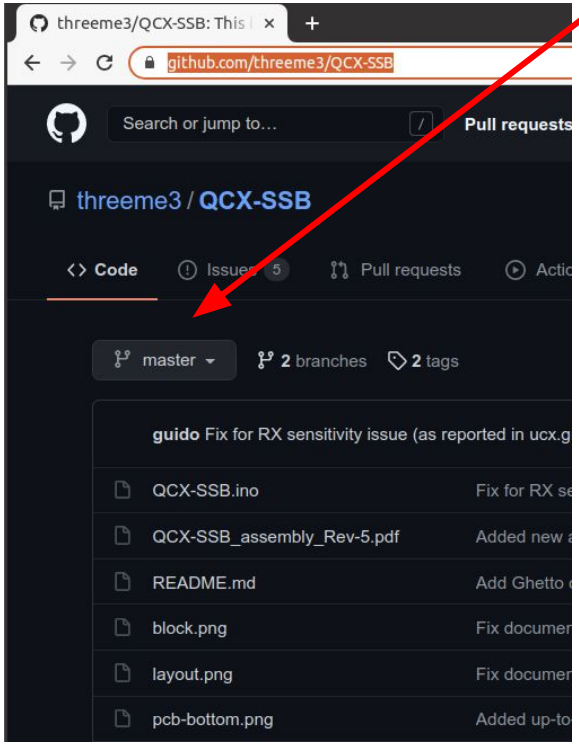

CLK0
CLK1
CLK2
Si5351
Clock Gen
8kHz-160 MHz
25MHz Xtal
SCL
SDA
GND
VIN
1602A
74ACT00
L-Button
R-Button

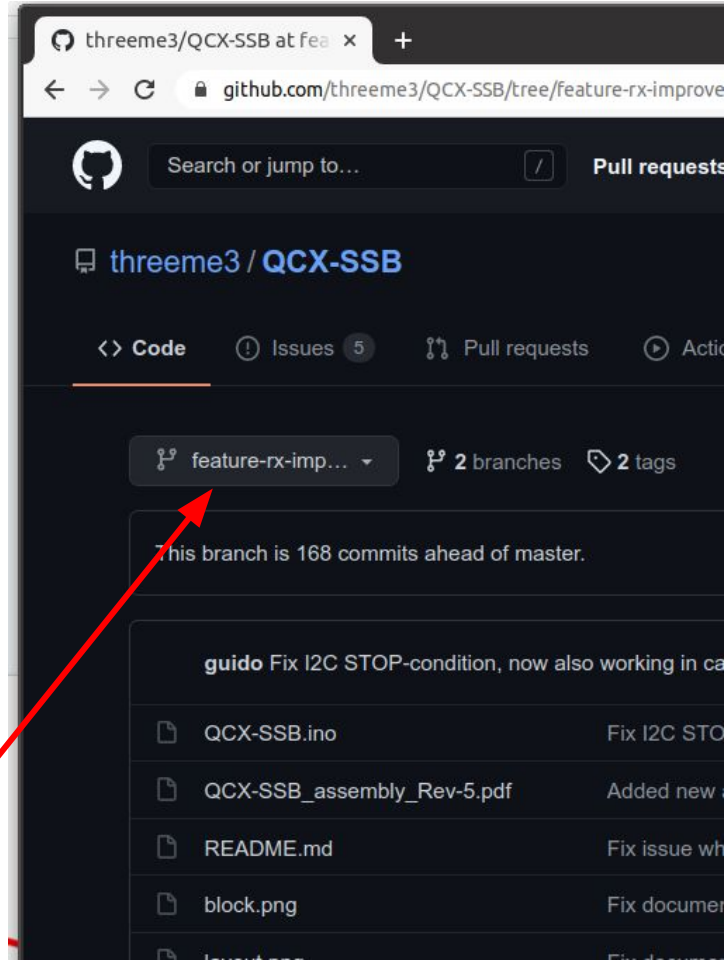
HATA MESAJLARI

HATA	PIN	DEGER
V5.0	5V Regulator	5V
V3.3	SCL (PDA4, PIN6)	3V2 - 3V8
Vavcc	AVCC (PIN20)	5V
Vadc2	PC2/A2 (PIN25)	1V8 - 3V2
Vadc0	AUDIO1, PC0/A0 (PIN23)	1V8 - 3V2
Vadc1	AUDIO2, PC1/A1 (PIN24)	1V8 - 3V2
i2cspeed	Si5351 I2C/LCD RS	
BER_i2c	Si5351 I2C/LCD RS	

YAZILIM BLOKLARINA HIZLICA BAKALIM

https://github.com/threeme3/QCX-SSB





https://github.com/threeme3/QCX-SSB/tree/feature-rx-improved

```
//  QCX-SSB.ino - https://github.com/threeme3/QCX-SSB
//
//  Copyright 2019, 2020   Guido PE1NNZ <pe1nnz@amsat.org>
//
//  Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated docum

#define VERSION   "1.02j"

#define QCX     1         // If you DO NOT have a QCX then comment-out (add two-slashes // in the beginning of this li

// QCX pin defintions
#define LCD_D4  0         //PD0    (pin 2)
#define LCD_D5  1         //PD1    (pin 3)
#define LCD_D6  2         //PD2    (pin 4)
#define LCD_D7  3         //PD3    (pin 5)
#define LCD_EN  4         //PD4    (pin 6)
#define FREQCNT 5         //PD5    (pin 11)
#define ROT_A   6         //PD6    (pin 12)
#define ROT_B   7         //PD7    (pin 13)
#define RX      8         //PB0    (pin 14)
#define SIDETONE 9        //PB1    (pin 15)
#define KEY_OUT 10        //PB2    (pin 16)
#define SIG_OUT 11        //PB3    (pin 17)
#define DAH     12        //PB4    (pin 18)
#define DIT     13        //PB5    (pin 19)
#define AUDIO1  14        //PC0/A0 (pin 23)
#define AUDIO2  15        //PC1/A1 (pin 24)
#define DVM     16        //PC2/A2 (pin 25)
#define BUTTONS 17        //PC3/A3 (pin 26)
#define LCD_RS  18        //PC4    (pin 27)
#define SDA     18        //PC4    (pin 27)
#define SCL     19        //PC5    (pin 28)
//#define NTX  11          //PB3    (pin 17)  - experimental: LOW on TX
```

LCD PIN TANIMLARI

```
class LCD : public Print {  // inspired by: http://www.technoblogy.com/show?2BET
public:  // LCD1602 display in 4-bit mode, RS is pull-up and kept low when idle to prevent potential display RFI via RS line
  #define _dn  0      // PD0 to PD3 connect to D4 to D7 on the display
  #define _en  4      // PC4 - MUST have pull-up resistor
  #define _rs  4      // PC4 - MUST have pull-up resistor
  //#define LCD_RS_HI() DDRC &= ~(1 << _rs);          // RS high (pull-up)
  //#define LCD_RS_LO() DDRC |= 1 << _rs;             // RS low (pull-down)
  #define LCD_RS_LO() PORTC &= ~(1 << _rs);         // RS low
  #define LCD_RS_HI() PORTC |= (1 << _rs);          // RS high
  #define LCD_EN_LO() PORTD &= ~(1 << _en);         // EN low
  #define LCD_EN_HI() PORTD |= (1 << _en);          // EN high
  #define LCD_PREP_NIBBLE(b) (PORTD & ~(0xf << _dn)) | (b) << _dn | 1 << _en // Send data and enable high
  void begin(uint8_t x = 0, uint8_t y = 0){         // Send command , make sure at least 40ms after power-up before sending commands
    bool reinit = (x == 0) && (y == 0);
    DDRD |= 0xf << _dn | 1 << _en;                  // Make data, EN outputs
    DDRC |= 1 << _rs;
    //PORTC &= ~(1 << _rs);                           // Set RS low in case to support pull-down when DDRC is output
    delayMicroseconds(50000);                       // *
    LCD_RS_LO(); LCD_EN_LO();
    cmd(0x33);                                      // Ensures display is in 8-bit mode
    delayMicroseconds(4500); cmd(0x33); delayMicroseconds(4500); cmd(0x33); delayMicroseconds(150); // * Ensures display is in 8-bit mode
    cmd(0x32);                                      // Puts display in 4-bit mode
    cmd(0x28);                                      // * Function set: 2-line, 5x8
    cmd(0x0c);                                      // Display on
    if(reinit) return;
    cmd(0x01);                                      // Clear display
    delay(3);                                       // Allow to execute Clear on display [https://www.sparkfun.com/datasheets/LCD/HD44780.
    cmd(0x06);                                      // * Entrymode: left, shift-dec
  }
  void nib(uint8_t b){                              // Send four bit nibble to display
    PORTD = LCD_PREP_NIBBLE(b);                     // Send data and enable high
    //asm("nop");                                     // Enable high pulse width must be at least 230ns high, data-setup time 80ns
    delayMicroseconds(4);
    LCD_EN_LO();
    //delayMicroseconds(52);                          // Execution time
    delayMicroseconds(60);                          // Execution time
  }
  void cmd(uint8_t b){ nib(b >> 4); nib(b & 0xf); }// Write command: send nibbles while RS low
  size_t write(uint8_t b){                          // Write data:    send nibbles while RS high
    //LCD_EN_HI();                                    // Complete Enable cycle must be at least 500ns (so start early)
    uint8_t nibh = LCD_PREP_NIBBLE(b >>  4);        // Prepare high nibble data and enable high
    PORTD = nibh;                                   // Send high nibble data and enable high
    uint8_t nibl = LCD_PREP_NIBBLE(b & 0xf);        // Prepare low nibble data and enable high
    //asm("nop");                                     // Enable high pulse width must be at least 230ns high, data-setup time 80ns; ATMEGA
    LCD_RS_HI();
    LCD_EN_LO();
    PORTD = nibl;                                   // Send low nibble data and enable high
    LCD_RS_LO();
    //asm("nop"); asm("nop");                         // Complete Enable cycle must be at least 500ns
    //PORTD = nibl;                                   // Send low nibble data and enable high
    //asm("nop");                                     // Enable high pulse width must be at least 230ns high, data-setup time 80ns; ATMEGA
    LCD_RS_HI();
    LCD_EN_LO();
    LCD_RS_LO();
    delayMicroseconds(60);                          // Execution time  (37+4)*1.25 us
    PORTD |= 0x02;                                  // To support serial-interface keep LCD_D5 high, so that DVM is not pulled-down via D
    return 1;
  }
  void setCursor(uint8_t x, uint8_t y){ cmd(0x80 | (x + y * 0x40)); }
  void cursor(){ cmd(0x0e); }
  void noCursor(){ cmd(0x0c); }
  void noDisplay(){ cmd(0x08); }
  void createChar(uint8_t l, uint8_t glyph[]){ cmd(0x40 | ((l & 0x7) << 3)); for(int i = 0; i != 8; i++) write(glyph[i]); }
};

/*
```

LCD FONKSIYONALRI

```
class LCD : public Print {  // inspired by: http://www.technoblogy.com/show?28ET
public:  // LCD1602 display in 4-bit mode, RS is pull-up and kept low when idle to prevent potential display RFI via RS line
  #define _dn  0      // PD0 to PD3 connect to D4 to D7 on the display
  #define _en  4      // PC4 - MUST have pull-up resistor
  #define _rs  4      // PC4 - MUST have pull-up resistor
  //#define LCD_RS_HI() DDRC &= ~(1 << _rs);          // RS high (pull-up)
  //#define LCD_RS_LO() DDRC |= 1 << _rs;             // RS low (pull-down)
  #define LCD_RS_LO() PORTC &= ~(1 << _rs);         // RS low
  #define LCD_RS_HI() PORTC |= (1 << _rs);          // RS high
  #define LCD_EN_LO() PORTD &= ~(1 << _en);         // EN low
  #define LCD_EN_HI() PORTD |= (1 << _en);          // EN high
  #define LCD_PREP_NIBBLE(b) (PORTD & ~(0xf << _dn)) | (b) << _dn | 1 << _en // Send data and enable high
  void begin(uint8_t x = 0, uint8_t y = 0){         // Send command , make sure at least 40ms after power-up before sending commands
    bool reinit = (x == 0) && (y == 0);
    DDRD |= 0xf << _dn | 1 << _en;                  // Make data, EN outputs
    DDRC |= 1 << _rs;
    //PORTC &= ~(1 << _rs);                           // Set RS low in case to support pull-down when DDRC is output
    delayMicroseconds(50000);                       // *
    LCD_RS_LO(); LCD_EN_LO();
    cmd(0x33);                                      // Ensures display is in 8-bit mode
    delayMicroseconds(4500); cmd(0x33); delayMicroseconds(4500); cmd(0x33); delayMicroseconds(150); // * Ensures display is in 8-bit mode
    cmd(0x32);                                      // Puts display in 4-bit mode
    cmd(0x28);                                      // * Function set: 2-line, 5x8
    cmd(0x0c);                                      // Display on
    if(reinit) return;
    cmd(0x01);                                      // Clear display
    delay(3);                                       // Allow to execute Clear on display [https://www.sparkfun.com/datasheets/LCD/HD44780.
    cmd(0x06);                                      // * Entrymode: left, shift-dec
  }
  void nib(uint8_t b){                              // Send four bit nibble to display
    PORTD = LCD_PREP_NIBBLE(b);                     // Send data and enable high
    //asm("nop");                                     // Enable high pulse width must be at least 230ns high, data-setup time 80ns
    delayMicroseconds(4);
    LCD_EN_LO();
    //delayMicroseconds(52);                          // Execution time
    delayMicroseconds(60);                          // Execution time
  }
  void cmd(uint8_t b){ nib(b >> 4); nib(b & 0xf); }// Write command: send nibbles while RS low
  size_t write(uint8_t b){                          // Write data:    send nibbles while RS high
    //LCD_EN_HI();                                    // Complete Enable cycle must be at least 500ns (so start early)
    uint8_t nibh = LCD_PREP_NIBBLE(b >>  4);        // Prepare high nibble data and enable high
    PORTD = nibh;                                   // Send high nibble data and enable high
    uint8_t nibl = LCD_PREP_NIBBLE(b & 0xf);        // Prepare low nibble data and enable high
    //asm("nop");                                     // Enable high pulse width must be at least 230ns high, data-setup time 80ns; ATMEGA
    LCD_RS_HI();
    LCD_EN_LO();
    PORTD = nibl;                                   // Send low nibble data and enable high
    LCD_RS_LO();
    //asm("nop"); asm("nop");                         // Complete Enable cycle must be at least 500ns
    //PORTD = nibl;                                   // Send low nibble data and enable high
    //asm("nop");                                     // Enable high pulse width must be at least 230ns high, data-setup time 80ns; ATMEGA
    LCD_RS_HI();
    LCD_EN_LO();
    LCD_RS_LO();
    delayMicroseconds(60);                          // Execution time  (37+4)*1.25 us
    PORTD |= 0x02;                                  // To support serial-interface keep LCD_D5 high, so that DVM is not pulled-down via D
    return 1;
  }
  void setCursor(uint8_t x, uint8_t y){ cmd(0x80 | (x + y * 0x40)); }
  void cursor(){ cmd(0x0e); }
  void noCursor(){ cmd(0x0c); }
  void noDisplay(){ cmd(0x08); }
  void createChar(uint8_t l, uint8_t glyph[]){ cmd(0x40 | ((l & 0x7) << 3)); for(int i = 0; i != 8; i++) write(glyph[i]); }
};

/*
```

LCD KOMUT FONKSIYONLARI

```
const uint8_t font[]PROGMEM = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x4f, 0x4f, 0x00, 0x00, 0x00,
    0x00, 0x07, 0x07, 0x00, 0x00, 0x07, 0x07, 0x00,
    0x14, 0x7f, 0x7f, 0x14, 0x14, 0x7f, 0x7f, 0x14,
    0x00, 0x24, 0x2e, 0x6b, 0x6b, 0x3a, 0x12, 0x00,
    0x00, 0x63, 0x33, 0x18, 0x0c, 0x66, 0x63, 0x00,
    0x00, 0x32, 0x7f, 0x4d, 0x4d, 0x77, 0x72, 0x50,
    0x00, 0x00, 0x00, 0x04, 0x06, 0x03, 0x01, 0x00,
    0x00, 0x00, 0x1c, 0x3e, 0x63, 0x41, 0x00, 0x00,
    0x00, 0x00, 0x41, 0x63, 0x3e, 0x1c, 0x00, 0x00,
    0x08, 0x2a, 0x3e, 0x1c, 0x1c, 0x3e, 0x2a, 0x08,
    0x00, 0x08, 0x08, 0x3e, 0x3e, 0x08, 0x08, 0x00,
    0x00, 0x00, 0x80, 0xe0, 0x60, 0x00, 0x00, 0x00,
    0x00, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x00,
    0x00, 0x00, 0x00, 0x60, 0x60, 0x00, 0x00, 0x00,
    0x00, 0x40, 0x60, 0x30, 0x18, 0x0c, 0x06, 0x02,
    0x00, 0x3e, 0x7f, 0x49, 0x45, 0x7f, 0x3e, 0x00,
    0x00, 0x40, 0x44, 0x7f, 0x7f, 0x40, 0x40, 0x00,
    0x00, 0x62, 0x73, 0x51, 0x49, 0x4f, 0x46, 0x00,
    0x00, 0x22, 0x63, 0x49, 0x49, 0x7f, 0x36, 0x00,
    0x00, 0x18, 0x18, 0x14, 0x16, 0x7f, 0x7f, 0x10,
    0x00, 0x27, 0x67, 0x45, 0x45, 0x7d, 0x39, 0x00,
    0x00, 0x3e, 0x7f, 0x49, 0x49, 0x7b, 0x32, 0x00,
    0x00, 0x03, 0x03, 0x79, 0x7d, 0x07, 0x03, 0x00,
    0x00, 0x36, 0x7f, 0x49, 0x49, 0x7f, 0x36, 0x00,
    0x00, 0x26, 0x6f, 0x49, 0x49, 0x7f, 0x3e, 0x00,
    0x00, 0x00, 0x00, 0x24, 0x24, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x80, 0xe4, 0x64, 0x00, 0x00, 0x00,
    0x00, 0x08, 0x1c, 0x36, 0x63, 0x41, 0x41, 0x00,
    0x00, 0x14, 0x14, 0x14, 0x14, 0x14, 0x14, 0x00,
    0x00, 0x41, 0x41, 0x63, 0x36, 0x1c, 0x08, 0x00,
    0x00, 0x02, 0x03, 0x51, 0x59, 0x0f, 0x06, 0x00,
    0x00, 0x3e, 0x7f, 0x41, 0x4d, 0x4f, 0x2e, 0x00,
    0x00, 0x7c, 0x7e, 0x0b, 0x0b, 0x7e, 0x7c, 0x00,
    0x00, 0x7f, 0x7f, 0x49, 0x49, 0x7f, 0x36, 0x00,
    0x00, 0x3e, 0x7f, 0x41, 0x41, 0x63, 0x22, 0x00,
    0x00, 0x7f, 0x7f, 0x41, 0x63, 0x3e, 0x1c, 0x00,
```

LCD
KARAKTERLERI

```
#define BRIGHT  1
static const uint8_t ssd1306_init_sequence [] PROGMEM = {  // Initialization Sequence
//  0xAE,     // Display OFF (sleep mode)
    0x20, 0b10,   // Set Memory Addressing Mode
          // 00=Horizontal Addressing Mode; 01=Vertical Addressing Mode;
          // 10=Page Addressing Mode (RESET); 11=Invalid
 0xB0,     // Set Page Start Address for Page Addressing Mode, 0-7
  0xC8,     // Set COM Output Scan Direction.  Flip Veritically.
 0x00,     // Set low nibble of column address
 0x10,     // Set high nibble of column address
   0x40,     // Set display start line address
#ifdef BRIGHT
  0x81, /*32*/ 0x7F,   // Set contrast control register
#else
  0x81, 32,   // Set contrast control register
#endif
  0xA1,     // Set Segment Re-map. A0=column 0 mapped to SEG0; A1=column 127 mapped to
   0xA6,     // Set display mode. A6=Normal; A7=Inverse
  0xA8, 0x1F,   // Set multiplex ratio(1 to 64)
   0xA4,     // Output RAM to Display
          // 0xA4=Output follows RAM content; 0xA5,Output ignores RAM content
  0xD3, 0x00,   // Set display offset. 00 = no offset
   0xD5, 0x80,   // --set display clock divide ratio/oscillator frequency
#ifdef BRIGHT
  0xD9, 0xF1, // 0xF1=brighter //0x22,   // Set pre-charge period
#else
  0xD9, 0x22,   // Set pre-charge period
#endif
  0xDA, 0x02,   // Set com pins hardware configuration
//   0xDB, 0x40, //0x20,   // --set vcomh 0x20 = 0.77xVcc
  0x8D, 0x14,    // Set DC-DC enable
  0xAF,     // Display ON
};

class SSD1306Device: public Print {
```

SSD1306 DSPLAY FONKSIYONLARI

```
class SSD1306Device: public Print {
public:
  #define SSD1306_ADDR 0x3C  // Slave address
  #define SSD1306_PAGES 4
  #define SSD1306_COMMAND 0x00
  #define SSD1306_DATA 0x40
  uint8_t oledX = 0, oledY = 0;
  uint8_t renderingFrame = 0xB0;
  bool wrap = false;

  void begin(uint8_t cols, uint8_t rows, uint8_t charsize = 0){
    Wire.begin();
    Wire.beginTransmission(SSD1306_ADDR); Wire.write(SSD1306_COMMAND);
    for (uint8_t i = 0; i < sizeof(ssd1306_init_sequence); i++) {
      Wire.write(pgm_read_byte(&ssd1306_init_sequence[i]));
    }
    Wire.endTransmission();
    delayMicroseconds(100);
  }
  void noCursor(){}
  void cursor(){}
  void noDisplay(){}
  void createChar(uint8_t l, uint8_t glyph[]){}

  void _setCursor(uint8_t x, uint8_t y) { oledX = x; oledY = y;
    Wire.beginTransmission(SSD1306_ADDR); Wire.write(SSD1306_COMMAND);
    Wire.write(renderingFrame | (oledY & 0x07));
    Wire.write(0x10 | ((oledX & 0xf0) >> 4));
    Wire.write(oledX & 0x0f);
    Wire.endTransmission();
  }
  void setCursor(uint8_t x, uint8_t y) { _setCursor(x * FONT_W, y * FONT_H); }

  void newLine() {
    oledY+=FONT_H;
    if (oledY > SSD1306_PAGES - FONT_H) {
      oledY = SSD1306_PAGES - FONT_H;
    }
    setCursor(0, oledY);
  }

  size_t write(byte c) {
    if((c == '\n') || (oledX > ((uint8_t)128 - FONT_W))) {
      if(wrap)  newLine();
      return 1;
    }
```

SD1306 FONKSIYONLARI

```
}
void encoder_setup()
{
  pinMode(ROT_A, INPUT_PULLUP);
  pinMode(ROT_B, INPUT_PULLUP);
  PCMSK2 |= (1 << PCINT22) | (1 << PCINT23); // interrupt-enable
  PCICR |= (1 << PCIE2);
  last_state = (digitalRead(ROT_B) << 1) | digitalRead(ROT_A);
  interrupts();
}
/*
class Encoder {
```

ENCODER

```
class I2C {
public:
  #define I2C_DELAY   4    // Determines I2C Speed (2=939kb/s (too fast!!); 3=822kb
  #define I2C_DDR DDRC     // Pins for the I2C bit banging
  #define I2C_PIN PINC
  #define I2C_PORT PORTC
  #define I2C_SDA (1 << 4) // PC4
  #define I2C_SCL (1 << 5) // PC5
  #define DELAY(n) for(uint8_t i = 0; i != n; i++) asm("nop");
  #define I2C_SDA_GET() I2C_PIN & I2C_SDA
  #define I2C_SCL_GET() I2C_PIN & I2C_SCL
  #define I2C_SDA_HI() I2C_DDR &= ~I2C_SDA;
  #define I2C_SDA_LO() I2C_DDR |=  I2C_SDA;
  #define I2C_SCL_HI() I2C_DDR &= ~I2C_SCL; DELAY(I2C_DELAY);
  #define I2C_SCL_LO() I2C_DDR |=  I2C_SCL; DELAY(I2C_DELAY);

  I2C(){
    I2C_PORT &= ~( I2C_SDA | I2C_SCL );
    I2C_SCL_HI();
    I2C_SDA_HI();
    suspend();
  }
  ~I2C(){
    I2C_PORT &= ~( I2C_SDA | I2C_SCL );
    I2C_DDR &= ~( I2C_SDA | I2C_SCL );
  }
  inline void start(){
    resume();  //prepare for I2C
    I2C_SCL_LO();
    I2C_SDA_HI();
  }
  inline void stop(){
    I2C_SCL_HI();
    I2C_SDA_HI();
    I2C_DDR &= ~(I2C_SDA | I2C_SCL); // prepare for a start: pull-up both SDA, SCL
    suspend();
  }
  #define SendBit(data, mask) \
    if(data & mask){ \
      I2C_SDA_HI();  \
    } else {         \
      I2C_SDA_LO();  \
    }              \
    I2C_SCL_HI();    \
    I2C_SCL_LO();
```

I2C FONKSIYONLARI

```
class SI5351 {
public:
  volatile int32_t _fout;
  volatile uint8_t _div;  // note: uint8_t asserts fout > 3.5MHz with R_DIV=1
  volatile uint16_t _msa128min512;
  volatile uint32_t _msb128;
  volatile uint8_t pll_regs[8];

  #define BB0(x) ((uint8_t)(x))           // Bash byte x of int32_t
  #define BB1(x) ((uint8_t)((x)>>8))
  #define BB2(x) ((uint8_t)((x)>>16))

  #define FAST __attribute__((optimize("Ofast")))

  #define F_XTAL 27005000            // Crystal freq in Hz, nominal frequency 27004300
  //#define F_XTAL 25004000          // Alternate SI clock
  //#define F_XTAL 20004000          // A shared-single 20MHz processor/pll clock
  volatile uint32_t fxtal = F_XTAL;

  inline void FAST freq_calc_fast(int16_t df)  // note: relies on cached variables: _msb128, _msa1
  {
    #define _MSC  0x80000  //0x80000: 98% CPU load   0xFFFFF: 114% CPU load
    uint32_t msb128 = _msb128 + ((int64_t)(_div * (int32_t)df) * _MSC * 128) / fxtal;

    //#define _MSC  0xFFFFF  // Old algorithm 114% CPU load, shortcut for a fixed fxtal=27e6
    //register uint32_t xmsb = (_div * (_fout + (int32_t)df)) % fxtal;  // xmsb = msb * fxtal/(128
    //uint32_t msb128 = xmsb * 5*(32/32) - (xmsb/32);  // msb128 = xmsb * 159/32, where 159/32 = 1

    //#define _MSC  (F_XTAL/128)  // 114% CPU load  perfect alignment
    //uint32_t msb128 = (_div * (_fout + (int32_t)df)) % fxtal;

    uint32_t msp1 = _msa128min512 + msb128 / _MSC;  // = 128 * _msa + msb128 / _MSC - 512;
    uint32_t msp2 = msb128 % _MSC;  // = msb128 - msb128/_MSC * _MSC;

    //pll_regs[0] = BB1(msc);  // 3 regs are constant
    //pll_regs[1] = BB0(msc);
    //pll_regs[2] = BB2(msp1);
    pll_regs[3] = BB1(msp1);
    pll_regs[4] = BB0(msp1);
    pll_regs[5] = ((_MSC&0xF0000)>>(16-4))|BB2(msp2); // top nibble MUST be same as top nibble of _
    pll_regs[6] = BB1(msp2);
    pll_regs[7] = BB0(msp2);
  }
```

SI5351 FONKSIYONLARI

```
void freq(uint32_t fout, uint8_t i, uint8_t q){  // Set a CLK0,1 to fout Hz with phase i, q
    uint8_t msa; uint32_t msb, msc, msp1, msp2, msp3p2;
    uint8_t rdiv = 0;             // CLK pin sees fout/(2^rdiv)
    if(fout < 500000){ rdiv = 7; fout *= 128; }; // Divide by 128 for fout 4..500kHz

    uint16_t d = (16 * fxtal) / fout;  // Integer part
    if(fout > 30000000) d = (34 * fxtal) / fout; // when fvco is getting too low (400 MHz)

    if( (d * (fout - 5000) / fxtal) != (d * (fout + 5000) / fxtal) ) d++; // Test if multiplier remains same for fre
    if(d % 2) d++;  // even numbers preferred for divider (AN619 p.4 and p.6)
    uint32_t fvcoa = d * fout;  // Variable PLLA VCO frequency at integer multiple of fout at around 27MHz*16 = 432M
    msa = fvcoa / fxtal;     // Integer part of vco/fxtal
    msb = ((uint64_t)(fvcoa % fxtal)*_MSC) / fxtal; // Fractional part
    msc = _MSC;

    msp1 = 128*msa + 128*msb/msc - 512;
    msp2 = 128*msb - 128*msb/msc * msc;    // msp3 == msc
    msp3p2 = (((msc & 0x0F0000) <<4) | msp2);  // msp3 on top nibble
    uint8_t pll_regs[8] = { BB1(msc), BB0(msc), BB2(msp1), BB1(msp1), BB0(msp1), BB2(msp3p2), BB1(msp2), BB0(msp2) }
    SendRegister(26+0*8, pll_regs, 8); // Write to PLLA
    SendRegister(26+1*8, pll_regs, 8); // Write to PLLB

    msa = fvcoa / fout;     // Integer part of vco/fout
    msp1 = (128*msa - 512) | (((uint32_t)rdiv)<<20);     // msp1 and msp2=0, msp3=1, not fractional
    uint8_t ms_regs[8] = {0, 1, BB2(msp1), BB1(msp1), BB0(msp1), 0, 0, 0};
    SendRegister(42+0*8, ms_regs, 8); // Write to MS0
    SendRegister(42+1*8, ms_regs, 8); // Write to MS1
    SendRegister(42+2*8, ms_regs, 8); // Write to MS2
    SendRegister(16+0, 0x0C|3|0x00);  // CLK0: 0x0C=PLLA local msynth; 3=8mA; 0x40=integer division; bit7:6=0->power
    SendRegister(16+1, 0x0C|3|0x00);  // CLK1: 0x0C=PLLA local msynth; 3=8mA; 0x40=integer division; bit7:6=0->power
    SendRegister(16+2, 0x2C|3|0x00);  // CLK2: 0x2C=PLLB local msynth; 3=8mA; 0x40=integer division; bit7:6=0->power
    SendRegister(165, i * msa / 90);  // CLK0: I-phase (on change -> Reset PLL)
    SendRegister(166, q * msa / 90);  // CLK1: Q-phase (on change -> Reset PLL)
    if(iqmsa != ((i-q)*msa/90)){ iqmsa = (i-q)*msa/90; SendRegister(177, 0xA0); } // 0x20 reset PLLA; 0x80 reset PLL
    SendRegister(3, 0b11111100);      // Enable/disable clock

    _fout = fout;  // cache
    _div = d;
    _msa128min512 = fvcoa / fxtal * 128 - 512;
    _msb128=((uint64_t)(fvcoa % fxtal)*_MSC*128) / fxtal;
}
```

FREKANS AYARLAMA

```
inline int16_t ssb(int16_t in)
{
  static int16_t dc;

  int16_t i, q;
  uint8_t j;
  static int16_t v[16];

  for(j = 0; j != 15; j++) v[j] = v[j + 1];

  dc += (in - dc) / 2;
  v[15] = in - dc;     // DC decoupling
  //dc = in;  // this is actually creating a high-pass (emphasis) filter

  i = v[7];
  q = ((v[0] - v[14]) * 2 + (v[2] - v[12]) * 8 + (v[4] - v[10]) * 21 + (v[6] - v[8]) * 15) / 128 + (v[6] - v[8]) / 2; // Hilbert transform, 40dB side-band re

  uint16_t _amp = magn(i, q);
  if(vox) _vox(_amp > vox_thresh);
  //_amp = (_amp > vox_thresh) ? _amp : 0;   // vox_thresh = 1 is a good setting

  _amp = _amp << (drive);
#ifdef CONSTANT_AMP
  if(_amp < 4 ){ amp = 0; return 0; } //hack: for constant amplitude cases, set drive=1 for good results
  //digitalWrite(RX, (_amp < 4)); // fast on-off switching for constant amplitude case
#endif
  _amp = ((_amp > 255) || (drive == 8)) ? 255 : _amp; // clip or when drive=8 use max output
  amp = (tx) ? lut[_amp] : 0;

  static int16_t prev_phase;
  int16_t phase = arctan3(q, i);

  int16_t dp = phase - prev_phase;  // phase difference and restriction
  //dp = (amp) ? dp : 0;  // dp = 0 when amp = 0
  prev_phase = phase;

  if(dp < 0) dp = dp + _UA; // make negative phase shifts positive: prevents negative frequencies and will reduce spurs on other sideband
#ifdef MAX_DP
  if(dp > MAX_DP){ // dp should be less than half unit-angle in order to keep frequencies below F_SAMP_TX/2
    prev_phase = phase - (dp - MAX_DP);  // substract restdp
    dp = MAX_DP;
  }
#endif
  if(mode == USB)
    return dp * ( F_SAMP_TX / _UA); // calculate frequency-difference based on phase-difference
  else
    return dp * (-F_SAMP_TX / _UA);
}
```

SSB

```
// This is the ADC ISR, issued with sample-rate via timer1 compb interrupt.
// It performs in real-time the ADC sampling, calculation of SSB phase-differences, calculation of SI5351 frequency registers and send the registers to SI5351
static int16_t _adc;
void dsp_tx()
{ // jitter dependent things first
#ifdef MULTI_ADC  // SSB with multiple ADC conversions:
  int16_t adc;                        // current ADC sample 10-bits analog input, NOTE: first ADCL, then ADCH
  adc = ADC;
  ADCSRA |= (1 << ADSC);
  //OCR1BL = amp;                       // submit amplitude to PWM register (actually this is done in advance (about 140us) of phase-change, so that phase-de
  si5351.SendPLLBRegisterBulk();      // submit frequency registers to SI5351 over 731kbit/s I2C (transfer takes 64/731 = 88us, then PLL-loopfilter probably
  OCR1BL = amp;                      // submit amplitude to PWM register (takes about 1/32125 = 31us+/-31us to propagate) -> amplitude-phase-alignment error is
  adc += ADC;
  //ADCSRA |= (1 << ADSC);  // causes RFI on QCX-SSB units (not on units with direct biasing); ENABLE this line when using direct biasing!!
  int16_t df = ssb(_adc >> MIC_ATTEN); // convert analog input into phase-shifts (carrier out by periodic frequency shifts)
  adc += ADC;
  ADCSRA |= (1 << ADSC);
  si5351.freq_calc_fast(df);          // calculate SI5351 registers based on frequency shift and carrier frequency
  adc += ADC;
  ADCSRA |= (1 << ADSC);
  _adc = (adc/4 - 512);
#else  // SSB with single ADC conversion:
  ADCSRA |= (1 << ADSC);    // start next ADC conversion (trigger ADC interrupt if ADIE flag is set)
  //OCR1BL = amp;                       // submit amplitude to PWM register (actually this is done in advance (about 140us) of phase-change, so that phase-de
  si5351.SendPLLBRegisterBulk();      // submit frequency registers to SI5351 over 731kbit/s I2C (transfer takes 64/731 = 88us, then PLL-loopfilter probably
  OCR1BL = amp;                       // submit amplitude to PWM register (takes about 1/32125 = 31us+/-31us to propagate) -> amplitude-phase-alignment error
  int16_t adc = ADC - 512; // current ADC sample 10-bits analog input, NOTE: first ADCL, then ADCH
  int16_t df = ssb(adc >> MIC_ATTEN);  // convert analog input into phase-shifts (carrier out by periodic frequency shifts)
  si5351.freq_calc_fast(df);          // calculate SI5351 registers based on frequency shift and carrier frequency
#endif

#ifdef CARRIER_COMPLETELY_OFF_ON_LOW
  if(OCR1BL == 0){ si5351.SendRegister(SI_CLK_OE, (amp) ? 0b11111011 : 0b11111111); } // experimental carrier-off for low amplitudes
#endif

  if(!mox) return;
  OCR1AL = (adc << (mox-1)) + 128;  // TX audio monitoring
}
```

TX

```
void dsp_tx_cw()
{ // jitter dependent things first
  OCR1BL = lut[255];

  process_minsky();
  OCR1AL = (p_sin >> (16 - volume)) + 128;
}
```

```
void dsp_tx_am()
{ // jitter dependent things first
  ADCSRA |= (1 << ADSC);    // start next ADC conversion (trigger ADC interrupt if ADIE flag is set)
  OCR1BL = amp;                        // submit amplitude to PWM register (actually this is done in advanc
  int16_t adc = ADC - 512; // current ADC sample 10-bits analog input, NOTE: first ADCL, then ADCH
  int16_t in = (adc >> MIC_ATTEN);
  in = in << (drive-4);
  //static int16_t dc;
  //dc += (in - dc) / 2;
  //in = in - dc;     // DC decoupling
  #define AM_BASE 32
  in=max(0, min(255, (in + AM_BASE)));
  amp=in;// lut[in];
}
```

```
static int32_t signal;
static int16_t avg = 0;
static int16_t maxpk=0;
static int16_t k0=0;
static int16_t k1=0;
static uint8_t sym;
static int16_t ta=0;
const char m2c[] PROGMEM = "**ETIANMSURWDKGOHVF*L*PJBXCYZQ**54S3***2**+***J16=/***H*7*G*8*90***********?_****\"**.****@***
static uint8_t nsamp=0;

char cw(int16_t in)
{
  char ch = 0;
  int i;
  signal += abs(in);
  #define OSR 64 // (8*FS/1000)
  if((nsamp % OSR) == 0){   // process every 8 ms
    nsamp=0;
    if(!signal) return ch;
    signal = signal / OSR;  //normalize
    maxpk = signal > maxpk ? signal : maxpk;
    #define RT  4
    if(signal>(maxpk/2) ){  // threshold: 3dB below max signal
      k1++;  // key on
      k0=0;
    } else {
      k0++;  // key off
      if(k0>0 && k1>0){        //symbol space
        if(k1>(ta/100)) ta=RT*ta/100+(100-RT)*k1;                  // go slower
        if(k1>(ta/600) && k1<(ta/300)) ta=(100-RT)*ta/100+RT*k1*3;  // go faster
        if(k1>(ta/600)) sym=(sym<<1)|(k1>(ta/200));        // dit (0) or dash (1)
        k1=0;
      }
      if(k0>=(ta/200) && sym>1){ //letter space
        if(sym<128) ch=/*m2c[sym]*/ pgm_read_byte_near(m2c + sym);
        sym=1;
      }
      if(k0>=(ta/67)){        //word space (67=1.5/100)
        ch = ' ';
        k0=-1000*(ta/100);           //delay word spaces
      }
    }
    avg = avg*99/100 + signal*1/100;
    maxpk = maxpk*99/100 + signal*1/100;
    signal = 0;
  }
  nsamp++;
  return ch;
}
```

```
//static uint32_t gain = 1024;
static int16_t gain = 1024;
inline int16_t process_agc(int16_t in)
{
  //int16_t out = ((uint32_t)(gain) >> 20) * in;
  //gain = gain + (1024 - abs(out) + 512);
  int16_t out = (gain >= 1024) ? (gain >> 10) * in : in;
  //if(gain >= 1024) out = (gain >> 10) * in;  // net gain >= 1
  //else if(gain >= 16) out = ((gain >> 4) * in) >> 6;  // net gain < 1
  //else out = (gain * in) >> 10;
  int16_t accum = (1 - abs(out >> 10));
  if((INT16_MAX - gain) > accum) gain = gain + accum;
  if(gain < 1) gain = 1;
  return out;
}

inline int16_t process_nr_old(int16_t ac)
{
  ac = ac >> (6-abs(ac));  // non-linear below amp of 6; to reduce noise
  ac = ac << 3;
  return ac;
}
```

```
// Non-recursive CIC Filter (M=2, R=4) implementation, so two-stages of (followed by down-sampling with fac
// H1(z) = (1 + z^-1)^2 = 1 + 2*z^-1 + z^-2 = (1 + z^-2) + (2) * z^-1 = FA(z) + FB(z) * z^-1;
// with down-sampling before stage translates into poly-phase components: FA(z) = 1 + z^-1, FB(z) = 2
// source: Lyons Understanding Digital Signal Processing 3rd edition 13.24.1
void sdr_rx()
{
  // process I for even samples  [75% CPU@R=4;Fs=62.5k] (excluding the Comb branch and output stage)
  ADMUX = admux[1];  // set MUX for next conversion
  ADCSRA |= (1 << ADSC);    // start next ADC conversion
  int16_t adc = ADC - 511; // current ADC sample 10-bits analog input, NOTE: first ADCL, then ADCH
  func_ptr = sdr_rx_q;    // processing function for next conversion
  sdr_rx_common();

  // Only for I: correct I/Q sample delay by means of linear interpolation
  static int16_t prev_adc;
  int16_t corr_adc = (prev_adc + adc) / 2;
  prev_adc = adc;
  adc = corr_adc;

  //static int16_t dc;
  //dc += (adc - dc) / 2;  // we lose LSB with this method
  //dc = (3*dc + adc)/4;
  //int16_t ac = adc - dc;     // DC decoupling
  int16_t ac = adc;

#ifdef AUTO_ADC_BIAS
  param_b = (7*param_b + adc)/8;
#endif
  int16_t ac2;
  static int16_t z1;
  if(rx_state == 0 || rx_state == 4){  // 1st stage: down-sample by 2
    static int16_t za1;
    int16_t _ac = ac + za1 + z1 * 2;           // 1st stage: FA + FB
    za1 = ac;
    static int16_t _z1;
    if(rx_state == 0){                  // 2nd stage: down-sample by 2
      static int16_t _za1;
      ac2 = _ac + _za1 + _z1 * 2;              // 2nd stage: FA + FB
      _za1 = _ac;
      {
        ac2 >>= att2;  // digital gain control
        // post processing I and Q (down-sampled) results
        static int16_t v[7];
        i = v[0]; v[0] = v[1]; v[1] = v[2]; v[2] = v[3]; v[3] = v[4]; v[4] = v[5]; v[5] = v[6]; v[6] = ac2;

        int16_t ac = i + qh;
        ac = slow_dsp(ac);

        // Output stage
        static int16_t ozd1, ozd2;
        if(_init){ ac = 0; ozd1 = 0; ozd2 = 0; _init = 0; } // hack: on first sample init accumulators of fu
#define SECOND_ORDER_DUC  1
#ifdef SECOND_ORDER_DUC
        int16_t od1 = ac - ozd1; // Comb section
        ocomb = od1 - ozd2;
        ozd2 = od1;
#else
        ocomb = ac - ozd1; // Comb section
#endif
        ozd1 = ac;
      }
    } else _z1 = _ac;
  } else z1 = ac;

  rx_state++;
}
```

```
void sdr_rx_q()
{
  // process Q for odd samples  [75% CPU@R=4;Fs=62.5k] (excluding the Comb branch and output stage)
  ADMUX = admux[0];  // set MUX for next conversion
  ADCSRA |= (1 << ADSC);    // start next ADC conversion
  int16_t adc = ADC - 511; // current ADC sample 10-bits analog input, NOTE: first ADCL, then ADCH
  func_ptr = sdr_rx;    // processing function for next conversion
#ifdef SECOND_ORDER_DUC
//  sdr_rx_common();  //necessary? YES!... Maybe NOT!
#endif

  //static int16_t dc;
  //dc += (adc - dc) / 2;  // we lose LSB with this method
  //dc = (3*dc + adc)/4;
  //int16_t ac = adc - dc;     // DC decoupling
  int16_t ac = adc;

#ifdef AUTO_ADC_BIAS
  param_c = (7*param_c + adc)/8;
#endif
  int16_t ac2;
  static int16_t z1;
  if(rx_state == 3 || rx_state == 7){  // 1st stage: down-sample by 2
    static int16_t za1;
    int16_t _ac = ac + za1 + z1 * 2;           // 1st stage: FA + FB
    za1 = ac;
    static int16_t _z1;
    if(rx_state == 7){                  // 2nd stage: down-sample by 2
      static int16_t _za1;
      ac2 = _ac + _za1 + _z1 * 2;              // 2nd stage: FA + FB
      _za1 = _ac;
      {
        ac2 >>= att2;  // digital gain control
        // Process Q (down-sampled) samples
        static int16_t v[14];
        q = v[7];
        qh = ((v[0] - ac2) + (v[2] - v[12]) * 4) / 64 + ((v[4] - v[10]) + (v[6] - v[8])) / 8 + ((v[4
        //qh = ((v[0] - ac2) * 2 + (v[2] - v[12]) * 8 + (v[4] - v[10]) * 21 + (v[6] - v[8]) * 15) /
        for(uint8_t j = 0; j != 13; j++) v[j] = v[j + 1]; v[13] = ac2;
      }
      rx_state = 0; return;
    } else _z1 = _ac;
  } else z1 = ac;

  rx_state++;
}
```

```
void sdr_rx()
{
  static int16_t ocomb;
  static int16_t qh;

  uint8_t b = !(rx_state & 0x01);
  rx* p = &rx_inst[b];
  uint8_t _rx_state;
  int16_t ac;
  if(b){  // rx_state == 0, 2, 4, 6 -> I-stage
    ADMUX = admux[1];  // set MUX for next conversion
    ADCSRA |= (1 << ADSC);    // start next ADC conversion
    ac = ADC - 512; // current ADC sample 10-bits analog input, NOTE: first ADCL, then ADCH

    //sdr_common
    static int16_t ozi1, ozi2;
    if(_init){ ocomb=0; ozi1 = 0; ozi2 = 0; } // hack
    // Output stage [25% CPU@R=4;Fs=62.5k]
    #define SECOND_ORDER_DUC 1
    #ifdef SECOND_ORDER_DUC
    ozi2 = ozi1 + ozi2;          // Integrator section
    #endif
    ozi1 = ocomb + ozi1;
    #ifdef SECOND_ORDER_DUC
    if(volume) OCR1AL = min(max((ozi2>>5) + 128, 0), 255);  //if(volume) OCR1AL = min(max((ozi2>>5) + ICR1L/2, 0), ICR1L);  // center and clip
    #else
    if(volume) OCR1AL = (ozi1>>5) + 128;
    //if(volume) OCR1AL = min(max((ozi1>>5) + 128, 0), 255);  //if(volume) OCR1AL = min(max((ozi2>>5) + ICR1L/2, 0), ICR1L);  // center and cli
    #endif
    // Only for I: correct I/Q sample delay by means of linear interpolation
    static int16_t prev_adc;
    int16_t corr_adc = (prev_adc + ac) / 2;
    prev_adc = ac;
    ac = corr_adc;
    _rx_state = ~rx_state;
  } else {
    ADMUX = admux[0];  // set MUX for next conversion
    ADCSRA |= (1 << ADSC);    // start next ADC conversion
    ac = ADC - 512; // current ADC sample 10-bits analog input, NOTE: first ADCL, then ADCH
    _rx_state = rx_state;
  }

  if(_rx_state & 0x02){  // rx_state == I: 0, 4  Q: 3, 7  1st stage: down-sample by 2
    int16_t _ac = ac + p->za1 + p->z1 * 2;          // 1st stage: FA + FB
    p->za1 = ac;
    if(_rx_state & 0x04){                  // rx_state == I: 0  Q:7   2nd stage: down-sample by 2
      int16_t ac2 = _ac + p->_za1 + p->_z1 * 2;             // 2nd stage: FA + FB
      p->_za1 = _ac;
      if(b){
        // post processing I and Q (down-sampled) results
        ac2 >>= att2;  // digital gain control
        // post processing I and Q (down-sampled) results
        static int16_t v[7];
        i = v[0]; v[0] = v[1]; v[1] = v[2]; v[2] = v[3]; v[3] = v[4]; v[4] = v[5]; v[5] = v[6]; v[6] = ac2;  // Delay to match Hilbert transfor

        int16_t ac = i + qh;
        ac = slow_dsp(ac);

        // Output stage
        static int16_t ozd1, ozd2;
        if(_init){ ac = 0; ozd1 = 0; ozd2 = 0; _init = 0; } // hack: on first sample init accumulators of further stages (to prevent instability
        #ifdef SECOND_ORDER_DUC
        int16_t od1 = ac - ozd1; // Comb section
        ocomb = od1 - ozd2;
        ozd2 = od1;
        #else
        ocomb = ac - ozd1; // Comb section
        #endif
        ozd1 = ac;
      } else {
        ac2 >>= att2;  // digital gain control
        // Process Q (down-sampled) samples
        static int16_t v[14];
        q = v[7];
        qh = ((v[0] - ac2) + (v[2] - v[12]) * 4) / 64 + ((v[4] - v[10]) + (v[6] - v[8])) / 8 + ((v[4] - v[10]) * 5 - (v[6] - v[8]) ) / 128 + (v
        //qh = ((v[0] - ac2) * 2 + (v[2] - v[12]) * 8 + (v[4] - v[10]) * 21 + (v[6] - v[8]) * 15) / 128 + (v[6] - v[8]) / 2; // Hilbert transfo
        for(uint8_t j = 0; j != 13; j++) v[j] = v[j + 1]; v[13] = ac2;
      }
    } else p->_z1 = _ac;
  } else p->z1 = ac;  // rx_state == I: 2, 6  Q: 1, 5

  rx_state++;
}
```

```
void adc_start(uint8_t adcpin, bool ref1v1, uint32_t fs)
{
#ifndef AUTO_ADC_BIAS
  DIDR0 |= (1 << adcpin); // disable digital input
#endif
  ADCSRA = 0;             // clear ADCSRA register
  ADCSRB = 0;             // clear ADCSRB register
  ADMUX = 0;              // clear ADMUX register
  ADMUX |= (adcpin & 0x0f);    // set analog input pin
  ADMUX |= ((ref1v1) ? (1 << REFS1) : 0) | (1 << REFS0);  // set AREF=1.1V (Internal ref); otherwise AR
  ADCSRA |= ((uint8_t)log2((uint8_t)(F_CPU / 13 / fs))) & 0x07;  // ADC Prescaler (for normal conversio
  //ADCSRA |= (1 << ADIE);  // enable interrupts when measurement complete
  ADCSRA |= (1 << ADEN);  // enable ADC
  //ADCSRA |= (1 << ADSC);  // start ADC measurements
#ifdef ADC_NR
//  set_sleep_mode(SLEEP_MODE_ADC);  // ADC NR sleep destroys the timer2 integrity, therefore Idle slee
  set_sleep_mode(SLEEP_MODE_IDLE);
  sleep_enable();
#endif
}
```

```
void start_rx()
{
  _init = 1;
  rx_state = 0;
  func_ptr = sdr_rx;  //enable RX DSP/SDR
  adc_start(2, true, F_ADC_CONV); admux[2] = ADMUX;
  if(dsp_cap == SDR){
    adc_start(0, !(att == 1)/*true*/, F_ADC_CONV); admux[0] = ADMUX;
    adc_start(1, !(att == 1)/*true*/, F_ADC_CONV); admux[1] = ADMUX;
  } else { // ANALOG, DSP
    adc_start(0, false, F_ADC_CONV); admux[0] = ADMUX; admux[1] = ADMUX;
  }
  timer1_start(F_SAMP_PWM);
  timer2_start(F_SAMP_RX);
  TCCR1A &= ~(1 << COM1B1); digitalWrite(KEY_OUT, LOW); // disable KEY_OUT PWM
}

void switch_rxtx(uint8_t tx_enable){
  tx = tx_enable;
  TIMSK2 &= ~(1 << OCIE2A);  // disable timer compare interrupt
  //delay(1);
  noInterrupts();
  if(tx_enable){
    switch(mode){
      case USB:
      case LSB: func_ptr = dsp_tx; break;
      case CW:  func_ptr = dsp_tx_cw; break;
      case AM:  func_ptr = dsp_tx_am; break;
      case FM:  func_ptr = dsp_tx_fm; break;
    }
  } else func_ptr = sdr_rx;
  if((!dsp_cap) && (!tx_enable) && vox)  func_ptr = dummy; //hack: for SSB mode, disable dsp_rx during vox mode ena
  interrupts();
  if(tx_enable) ADMUX = admux[2];
  else _init = 1;
  rx_state = 0;
  if(tx_enable){
      digitalWrite(RX, LOW);  // TX (disable RX)
#ifdef NTX
      digitalWrite(NTX, LOW);  // TX (enable TX)
#endif
      lcd.setCursor(15, 1); lcd.print("T");
      si5351.SendRegister(SI_CLK_OE, 0b11111011); // CLK2_EN=1, CLK1_EN,CLK0_EN=0
      //if(!mox) TCCR1A &= ~(1 << COM1A1); // disable SIDETONE, prevent interference during TX
      OCR1AL = 0; // make sure SIDETONE is set to 0%
      TCCR1A |= (1 << COM1B1);  // enable KEY_OUT PWM
  } else {
      //TCCR1A |= (1 << COM1A1);  // enable SIDETONE
      TCCR1A &= ~(1 << COM1B1); digitalWrite(KEY_OUT, LOW); // disable KEY_OUT PWM, prevents interference during RX
      OCR1BL = 0; // make sure PWM (KEY_OUT) is set to 0%
      digitalWrite(RX, !(att == 2)); // RX (enable RX when attenuator not on)
#ifdef NTX
      digitalWrite(NTX, HIGH);  // RX (disable TX)
#endif
      si5351.SendRegister(SI_CLK_OE, 0b11111100); // CLK2_EN=0, CLK1_EN,CLK0_EN=1
      lcd.setCursor(15, 1); lcd.print((vox) ? "V" : "R");
  }
  OCR2A = (((float)F_CPU / (float)64) / (float)((tx_enable) ? F_SAMP_TX : F_SAMP_RX) + 0.5) - 1;
  TIMSK2 |= (1 << OCIE2A);  // enable timer compare interrupt TIMER2_COMPA_vect
}
```

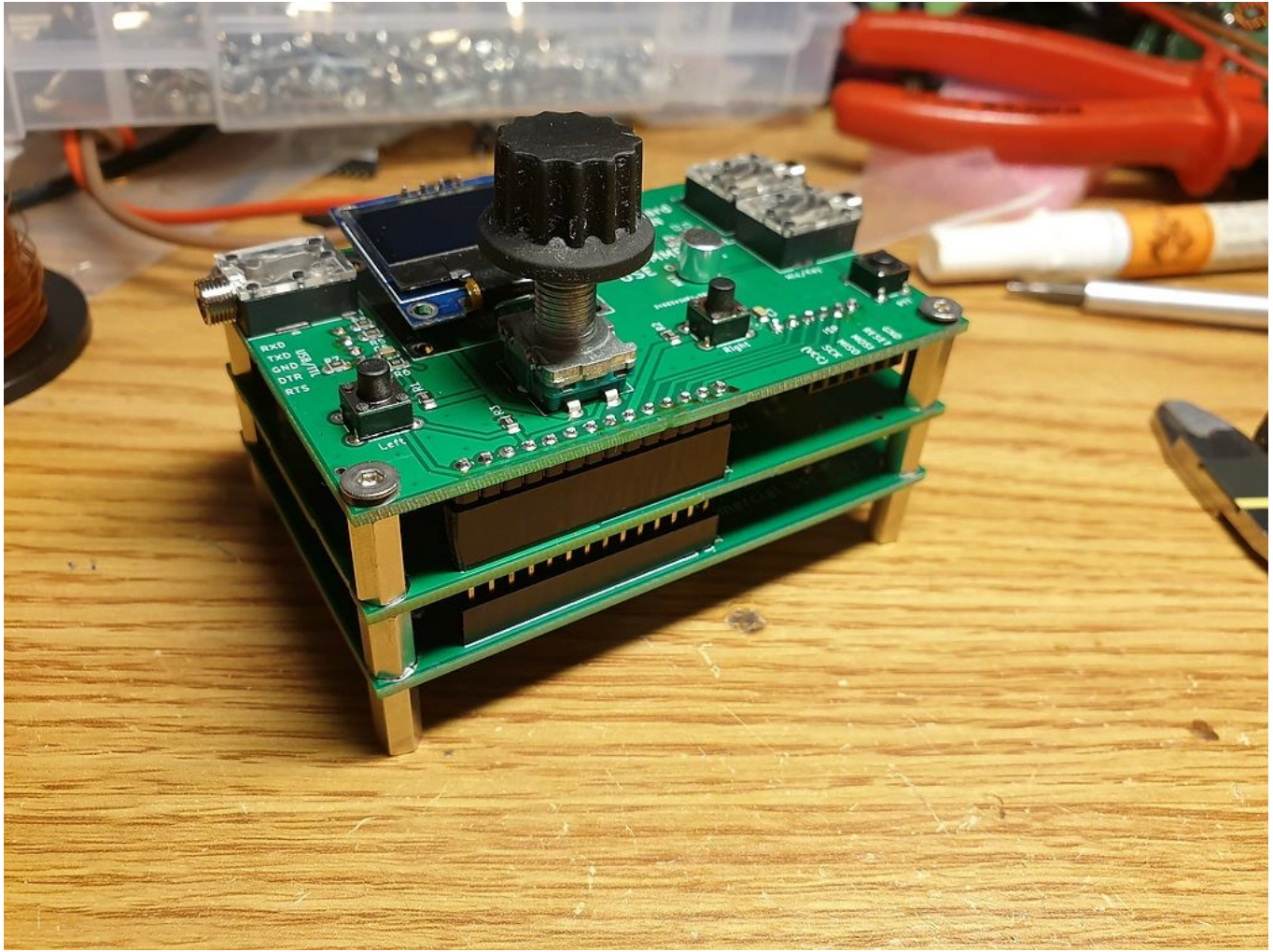
```
int16_t cal_iq_dummy = 0;
// RX I/Q calibration procedure: terminate with 50 ohm, enable CW filter, adjust R27, R24, R17 subsequently to its minimum side-band rejec
void calibrate_iq()
{
  smode = 1;
  lcd.setCursor(0, 0); lcd.print(blanks); lcd.print(blanks);
  digitalWrite(SIG_OUT, true); // loopback on
  si5351.freq(freq, 0, 90);  // RX in USB
  si5351.SendRegister(SI_CLK_OE, 0b11111000); // CLK2_EN=0, CLK1_EN,CLK0_EN=1
  float dbc;
  si5351.freq_calc_fast(+700); si5351.SendPLLBRegisterBulk(); delay(100);
  dbc = smeter();
  si5351.freq_calc_fast(-700); si5351.SendPLLBRegisterBulk(); delay(100);
  lcd.setCursor(0, 1); lcd.print("I-Q bal. 700Hz"); lcd.print(blanks);
  for(; !digitalRead(BUTTONS);){ wdt_reset(); smeter(dbc); } for(; digitalRead(BUTTONS);) wdt_reset();
  si5351.freq_calc_fast(+600); si5351.SendPLLBRegisterBulk(); delay(100);
  dbc = smeter();
  si5351.freq_calc_fast(-600); si5351.SendPLLBRegisterBulk(); delay(100);
  lcd.setCursor(0, 1); lcd.print("Phase Lo 600Hz"); lcd.print(blanks);
  for(; !digitalRead(BUTTONS);){ wdt_reset(); smeter(dbc); } for(; digitalRead(BUTTONS);) wdt_reset();
  si5351.freq_calc_fast(+800); si5351.SendPLLBRegisterBulk(); delay(100);
  dbc = smeter();
  si5351.freq_calc_fast(-800); si5351.SendPLLBRegisterBulk(); delay(100);
  lcd.setCursor(0, 1); lcd.print("Phase Hi 800Hz"); lcd.print(blanks);
  for(; !digitalRead(BUTTONS);){ wdt_reset(); smeter(dbc); } for(; digitalRead(BUTTONS);) wdt_reset();

  lcd.setCursor(9, 0); lcd.print(blanks);  // cleanup dbmeter
  digitalWrite(SIG_OUT, false); // loopback off
  si5351.SendRegister(SI_CLK_OE, 0b11111100); // CLK2_EN=0, CLK1_EN,CLK0_EN=1
  change = true;  //restore original frequency setting
}
#endif
#endif //QCX
```

```
void show_banner(){
  lcd.setCursor(0, 0);
#ifdef QCX
  lcd.print(F("QCX"));
  const char* cap_label[] = { "SSB", "DSP", "SDR" };
  if(ssb_cap || dsp_cap){ lcd.print(F("-")); lcd.print(cap_la
#else
  lcd.print(F("uSDX"));
#endif
  lcd.print(F("\x01 ")); lcd_blanks();
}

const char* mode_label[5] = { "LSB", "USB", "CW ", "AM ", "FM

void display_vfo(uint32_t f){
  lcd.setCursor(0, 1);
  lcd.print('\x06');  // VFO A/B

  uint32_t scale=10e6;  // VFO frequency
  if(f/scale == 0){ lcd.print(' '); scale/=10; }  // Initial
  for(; scale!=1; f%=scale, scale/=10){
    lcd.print(f/scale);
    if(scale == (uint32_t)1e3 || scale == (uint32_t)1e6) lcd.
  }

  lcd.print(" "); lcd.print(mode_label[mode]); lcd.print("  "
  lcd.setCursor(15, 1); lcd.print("R");
}
```

Revision History:

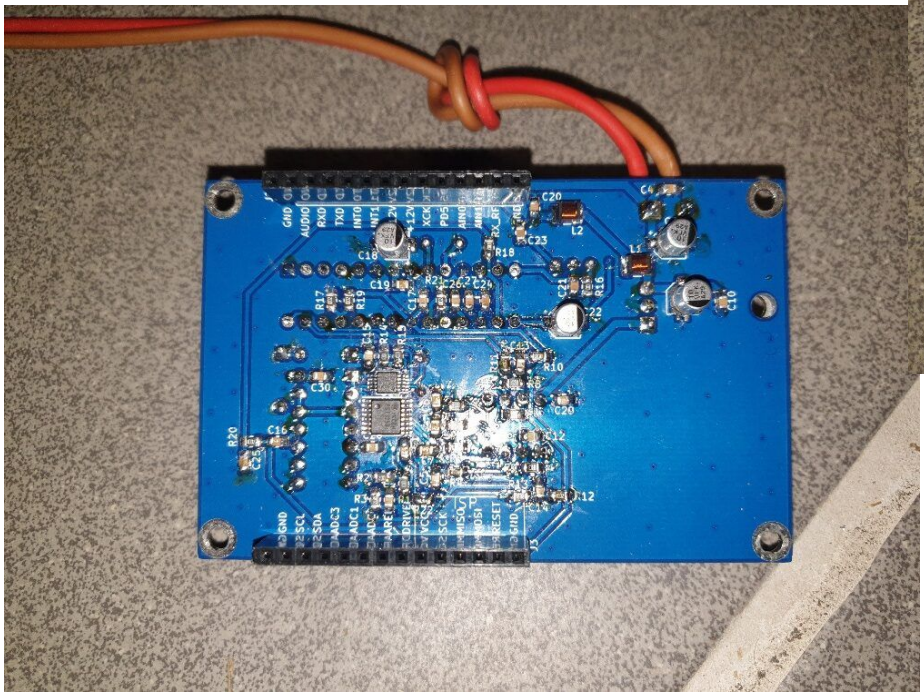
Rev.	Date	Features
[R1.02r]	2021-04-05	TX quality improvements, better robustness against RFI feedback, fix VOX issue, single encoder/button-only control option, 16MHz Arduino Uno/Nano support, CW Messages.
R1.02n	2021-02-22	Key click reduction, TX bandwidth control, OLED fixes, CAT remote control features including RX audio streaming.
R1.02m	2021-01-27	CW support, TS480 CAT support, RX quality improvments, semi-QSK, PA PTT out with TX-delay, VFO-A/B/RIT, LPF switching, backlight saving, 160m.
R1.02j	2020-10-10	Integrated SDR receiver, CW decoder, DSP filters, AGC, NR, ATT, experimental modes CW, AM, FM, quick menu, persistent settings, improved SSB TX quality. LCD fix, selectable CW pitch.
R1.01d	2019-05-05	Q6 now digitally switched (remove C31) - improving stability and IMD. Improved signal processing, audio quality, increased bandwidth, cosmetic changes and reduced RF feedback, reduced s-meter RFI, S-meter readings, self-test on startup. Receiver I/Q calibration, (experimental) amplitude pre-distortion and calibration. (Original QCX-SSB mod is described here R1.01d)
R1.00	2019-01-29	Initial release of SSB transceiver prototype.

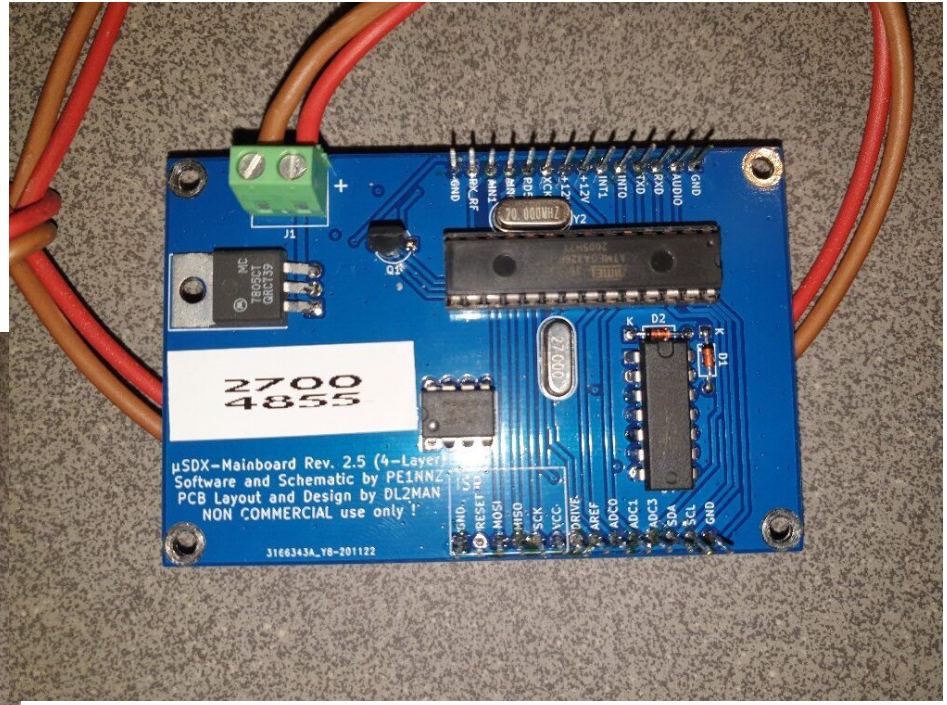
DL2MAN

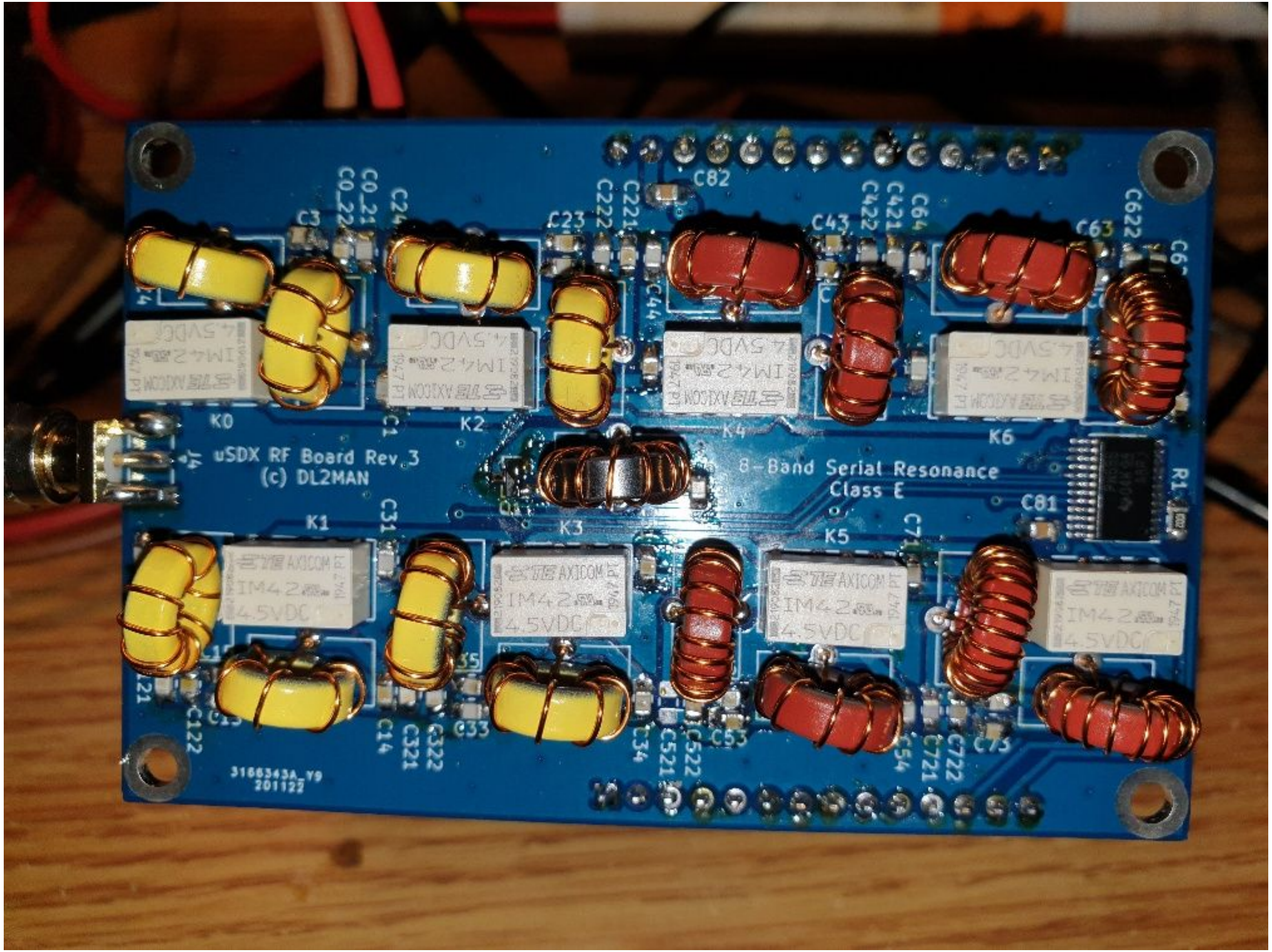



µSDX
OLED IO Board
by DL2MAN
V2.0


uSDX DL2MAN S2
A 7,100,00 LSB R
µSDX -IO
by DL2MAN
MK1
Mic
PTT
EMG Key
Left
Right




2700
4855
µSDX–Mainboard Rev. 2.5 (4–Layer)
Software and Schematic by PE1NNZ
PCB Layout and Design by DL2MAN
NON COMMERCIAL use only !
31E6343A_Y8-201122
20.000MHz
J1
Y2
Q1
D1
D2


uSDX RF Board Rev.3
(c) DL2MAN
8-Band Serial Resonance
Class E

TESEKKURLER