```
//  QCX-SSB.ino - https://github.com/threeme3/QCX-SSB
//
//  Copyright 2019, 2020, 2021   Guido PE1NNZ <pe1nnz@amsat.org>
//
//  Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in the
Software without restriction, including without limitation the rights to use, copy,
modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and
to permit persons to whom the Software is furnished to do so, subject to the following
conditions: The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS",
WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT
OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
//
//  Version modificada y castellanizada para la placa usdx_SD_V3, usdx_SD_V4 de EA2EHC.
//  Capacidades aÃ±adas para el aprendizaje y practica de CW (En proyecto y evolucion)
// -- Especial para el Grupo Tortugas --

#define VERSION   "1.02t"
// Interruptores de configuracion: Quita o pon la doble barra para activar o desactivar
diferentes caracteristicas.
#define KEYER                 1     // CW keyer
#define CW_DECODER            1     // Decodificador CW
//#define CW_LEARN            1     // Aprender y practiar CW metodo Koch, necesita
revison, No activar.
#define FILTER_700HZ          1     // Activa la opcion de 600Hz / 700Hz
//#define CAT                 1   // Interface CAT, usar emulacion del Kenwood TS-480
//#define CAT_EXT             1   // Soporte CAT extendido: remote button and screen
control commands over CAT
//#define CAT_STREAMING       1   // Soporte CAT extendido: audio streaming sobre CAT,
una vez activado y selecionado mediante el CAT cmd, 7.812ksps 8-bit unsigned audio se
envia mediante la  UART. El punto y como ";" es omitido en la trama de datos, y solo se
envia para indicar el principio y final del CAT cmd.
#define SWAP_ROTARY           1   // Cambia la direccion del encoder
#define KEY_CLICK        1   // Reduce los clicks del manipulador en la forma de onda.
#define SEMI_QSK         1   // En CW justo despues de manipular, mantiene la recepcion
muda durante un corto periodo de tiempo. No hay recepcion entre caracteres en  TX CW
#define RIT_ENABLE       1   // Se aplica un desplazamiento en RX respecto a TX
#define VOX_ENABLE       1   // Activa el trasmisor mediante un nivel de voz o digital
en la entrada de audio. El umbral de activacion se define en el menu 3.2
#define MOX_ENABLE       1   // Monitoriza a traves del altavoz la seÃ±al de audio
durante la transmision
#define FAST_AGC         1   // AÃ±ade la opcion de control automatico de ganancia
rapido, epecial para CW
//#define TX_DELAY         1   // AÃ±ade una temporizacion a la transmision por si se
usa un amplificador lineal externo conmutarlo primero mediante su entrada de rele.
//#define TUNING_DIAL      1   // Escanea la frecuencia mediante pulsacion larga del
pulsador central
#define SI5351_ADDR   0x60   // Direccion I2C del modulo SI5351A
#define F_XTAL  25001180       // Ajuste fino de la frecuencia del cristal del SI5351
#define F_MCU   16000000       // Frecuencia del cristal del arduino ATMEGA328P
#define CW_MESSAGE       1   // Mensaje de llamada CQ en CW. Se lanza haciendo click
izuierdo en el menu 4.2
#define CW_MESSAGE_EXT   1   // Mensajes adicionales CW para QSO en automatico
#define CW_FREQS_QRP     1   // Frecuencia por defecto CW QRP cuando cambiamos de banda
//#define CW_FREQS_FISTS   1   // Frecuencia por defecto CW VERTICAL cuando cambiamos
de banda

// Definicion de los pines del Arduino
#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, used as PTT
out to enable external PAs
//#define PTX     11        //PB3    (pin 17)  - experimental: HIGH on TX, used as PTT
out to enable external PAs

#ifdef SWAP_ROTARY
#undef ROT_A
#undef ROT_B
#define ROT_A   7         //PD7    (pin 13)
#define ROT_B   6         //PD6    (pin 12)
#endif

#if defined(CAT)
#define _SERIAL  1       // Coexistence support for serial port and LCD on the same
pins
#endif

#ifdef CW_LEARN
char letra;
int GROUP_NUM = 1;       // tren de caracteres que se mandan al estudiante cada vez de
uno en uno, de dos en dos ...
int GROUP_DLY = 30;      // tiempo entre caracteres enviados
int KOCH_NUM = 5;        // Numero de caracteres distintos que se mandan k, m, r, s , u
...
int KOCH_SKIP = 0;      // caracteres que no se envian: 1 no se envia k, 2 no se envia
m ...
int learn_mode = 0;
byte loop1 = 0;
byte kn = 0;
byte gx = 0;
byte ge = 0;
String cw_to_send_to_user(7);
String user_sent_cw = "";
#endif



//FUSES = { .low = 0xFF, .high = 0xD6, .extended = 0xFD };   // Fuse settings should be
set at programming (Arduino IDE > Tools > Burn bootloader)

#if(ARDUINO < 10810)
   #error "Unsupported Arduino IDE version, use Arduino IDE 1.8.10 or later from
https://www.arduino.cc/en/software"
#endif
#if !(defined(ARDUINO_ARCH_AVR))
   #error "Unsupported architecture, select Arduino IDE > Tools > Board > Arduino AVR
Boards > Arduino Uno."
#endif
#if(F_CPU != 16000000)
   #error "Unsupported clock frequency, Arduino IDE must specify 16MHz clock; alternate
crystal frequencies may be specified with F_MCU."
#endif
#undef F_CPU
#define F_CPU 20007000  // Actual crystal frequency of 20MHz XTAL1, note that this
declaration is just informative and does not correct the timing in Arduino functions
```

```
like delay(); hence a 1.25 factor needs to be added for correction.
#ifndef F_MCU
#define F_MCU 20000000  // 20MHz ATMEGA328P crystal
#endif

extern char __bss_end;
static int freeMemory(){ char* sp = reinterpret_cast<char*>(SP); return sp -
&__bss_end; }  // see: http://www.nongnu.org/avr-libc/user-manual/malloc.html

#ifdef CAT_EXT
volatile uint8_t cat_key = 0;
uint8_t _digitalRead(uint8_t pin){  // reads pin or (via CAT) artificially overriden
pins
  serialEvent();  // allows CAT update
  if(cat_key){ return (pin == BUTTONS) ? ((cat_key&0x07) > 0) : (pin == DIT) ?
~cat_key&0x10 : (pin == DAH) ? ~cat_key&0x20 : 0; } // overrides digitalRead(DIT, DAH,
BUTTONS);
  return digitalRead(pin);
}
#else
#define _digitalRead(x) digitalRead(x)
#endif //CAT_EXT



//#ifdef KEYER
// Iambic Morse Code Keyer Sketch, Contribution by Uli, DL2DBG. Copyright (c) 2009
Steven T. Elliott Source: http://openqrp.org/?p=343,  Trimmed by Bill Bishop -
wrb[at]wrbishop.com.  This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public License as published by the
Free Software Foundation; either version 2.1 of the License, or (at your option) any
later version. This library is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for more details:
Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
USA.

// keyerControl bit definitions
#define DIT_L    0x01     // Dit latch
#define DAH_L    0x02     // Dah latch
#define DIT_PROC 0x04     // Dit is being processed
#define PDLSWAP  0x08     // 0 for normal, 1 for swap
#define IAMBICB  0x10     // 0 for Iambic A, 1 for Iambic B
#define IAMBICA  0x00     // 0 for Iambic A, 1 for Iambic B
#define SINGLE   2        // Keyer Mode 0 1 -> Iambic2  2 ->SINGLE

int keyer_speed = 18;
static unsigned long ditTime;                    // No. milliseconds per dit
static uint8_t keyerControl;
static uint8_t keyerState;
static uint8_t keyer_mode = 2; //->  SINGLE
static uint8_t keyer_swap = 0; //->  DI/DAH

static uint32_t ktimer;
static int Key_state;
int debounce;

enum KSTYPE {IDLE, CHK_DIT, CHK_DAH, KEYED_PREP, KEYED, INTER_ELEMENT }; // State
machine states

void update_PaddleLatch() // Latch dit and/or dah press, called by keyer routine
{
    if(_digitalRead(DIT) == LOW) {
        keyerControl |= keyer_swap ? DAH_L : DIT_L;
    }
    if(_digitalRead(DAH) == LOW) {
        keyerControl |= keyer_swap ? DIT_L : DAH_L;
    }
}
```

```
void loadWPM (int wpm) // Calculate new time constants based on wpm value
{
#if(F_MCU != 20000000)
  ditTime = (1200ULL * F_MCU/16000000)/wpm;   //ditTime = 1200/wpm;  compensated for
F_CPU clock (running in a 16MHz Arduino environment)
#else
  ditTime = (1200 * 5/4)/wpm;   //ditTime = 1200/wpm;  compensated for 20MHz clock
(running in a 16MHz Arduino environment)
#endif
}
//#endif //KEYER
static uint8_t practice = false;  // Practice mode

volatile uint8_t cat_active = 0;
volatile uint32_t rxend_event = 0;
volatile uint8_t vox = 0;

#include <avr/sleep.h>
#include <avr/wdt.h>

uint8_t backlight = 8;

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      // PD4 - MUST have pull-up resistor
  #define _rs  4      // PC4 - MUST have pull-up resistor
#define RS_PULLUP  1   // Use pullup on RS line, ensures compliancy to the absolute
maximum ratings for the si5351 sda input that is shared with rs pin of lcd
#ifdef RS_PULLUP
  #define LCD_RS_HI() DDRC &= ~(1 << _rs); asm("nop"); // RS high (pull-up)
  #define LCD_RS_LO() DDRC |= 1 << _rs;                // RS low (pull-down)
#else
  #define LCD_RS_LO() PORTC &= ~(1 << _rs);        // RS low
  #define LCD_RS_HI() PORTC |= (1 << _rs);         // RS high
#endif //RS_PULLUP
  #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.pdf, p.49, p58]
    cmd(0x06);                                     // * Entrymode: left, shift-dec
  }
  void nib(uint8_t b){                             // Send four bit nibble to display
    pre();
    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();
    post();
```

```
    //delayMicroseconds(52);                          // Execution time
    delayMicroseconds(60);                          // Execution time
  }
  // Since LCD is using PD0(RXD), PD1(TXD) pins in the data-path, some co-existence
feature is required when using the serial port.
  // The following functions are temporarily dsiabling the serial port when LCD writes
happen, and make sure that serial transmission is ended.
  // To prevent that LCD writes are received by the serial receiver, PC2 is made HIGH
during writes to pull-up TXD via a diode.
  // The RXD, TXD lines are connected to the host via 1k resistors, a 1N4148 is placed
between PC2 (anode) and the TXD resistor.
  // There are two drawbacks when continuous LCD writes happen: 1. noise is leaking via
the AREF pull-ups into the receiver 2. serial data cannot be received.
  void pre(){
#ifdef _SERIAL
    if(!vox) if(cat_active){ Serial.flush(); for(; millis() < rxend_event;)wdt_reset();
PORTC |= 1<<2; DDRC |= 1<<2; } UCSR0B &= ~((1<<RXEN0)|(1<<TXEN0)); // Complete serial
TX and RX; mask PD1 LCD data-exchange by pulling-up TXD via PC2 HIGH; enable PD0/PD1,
disable serial port
#endif
    noInterrupts();  // do not allow LCD tranfer to be interrupted, to prevent
backlight to lighten-up
  }
  void post(){
    if(backlight) PORTD |= 0x08; else PORTD &= ~0x08;   // Backlight control
#ifdef _SERIAL
    //UCSR0B |= (1<<RXEN0)|(1<<TXEN0); if(!vox) if(cat_active){ DDRC &= ~(1<<2); } //
Enable serial port, disable PD0, PD1; disable PC2
    UCSR0B |= (1<<RXEN0)|(1<<TXEN0); if(!vox) if(cat_active){ PORTC &= ~(1<<2); } //
Enable serial port, disable PD0, PD1; PC2 LOW to prevent CAT TX disruption via MIC
input
#endif
    interrupts();
  }
  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
    pre();
    //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 clock-cycle is 50ns (so at least 5
cycles)
    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 clock-cycle is 50ns (so at least 5
cycles)
    LCD_RS_HI();
    LCD_EN_LO();
    LCD_RS_LO();
    post();
    delayMicroseconds(60);                          // Execution time  (37+4)*1.25 us
    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]); }
};



template<class parent>class Display : public parent {  // This class spoofs display
contents and cursor state
public:
#ifdef CAT_EXT
  uint8_t x, y;
  bool curs;
  char text[2*16+1];
  Display() : parent() { clear(); };
  size_t write(uint8_t b){ if((x<16) && (y<2)){ text[y*16+x] = ((b < 9) ? ">
:*#AB"[b-1] /*(b + 0x80 - 1)*/ /*(uint8_t[]){ 0xAF, 0x20, 0xB0, 0xB1, 0xB2, 0xA6, 0xE1,
0x20 }[b-1]*/ : b); x++; } return parent::write(b); }
  void setCursor(uint8_t _x, uint8_t _y){ x = _x; y = _y; parent::setCursor(_x, _y); }
  void cursor(){ curs = true; parent::cursor(); }
  void noCursor(){ curs = false; parent::noCursor(); }
  void clear(){ for(uint8_t i = 0; i != 2*16; i++) text[i] = ' ' ; text[2*16] = '\0'; x
= 0; y = 0; }
#endif //CAT_EXT
};

Display<LCD> lcd;     // highly-optimized LCD driver, OK for QCX supplied displays

volatile int8_t encoder_val = 0;
volatile int8_t encoder_step = 0;
static uint8_t last_state;
ISR(PCINT2_vect){  // Interrupt on rotary encoder turn
  //noInterrupts();
  //PCMSK2 &= ~((1 << PCINT22) | (1 << PCINT23));  // mask ROT_A, ROT_B interrupts
  switch(last_state = (last_state << 4) | (_digitalRead(ROT_B) << 1) |
_digitalRead(ROT_A)){ //transition  (see: https://www.allaboutcircuits.com/projects
/how-to-use-a-rotary-encoder-in-a-mcu-based-project/  )
//#define ENCODER_ENHANCED_RESOLUTION  1
#ifdef ENCODER_ENHANCED_RESOLUTION // Option: enhance encoder from 24 to 96
steps/revolution, see: appendix 1, https://www.sdr-kits.net/documents/PA0KLT_Manual.pdf
    case 0x31: case 0x10: case 0x02: case 0x23: encoder_val++; break;
    case 0x32: case 0x20: case 0x01: case 0x13: encoder_val--; break;
#else
    case 0x23:  encoder_val++; break;
    case 0x32:  encoder_val--; break;
#endif
  }
}
void encoder_setup()
{
  pinMode(ROT_A, INPUT_PULLUP);
  pinMode(ROT_B, INPUT_PULLUP);
  PCMSK2 |= (1 << PCINT22) | (1 << PCINT23); // interrupt-enable for ROT_A, ROT_B pin
changes; see https://github.com/EnviroDIY/Arduino-SDI-12/wiki/2b.-Overview-of-
Interrupts
  PCICR |= (1 << PCIE2);
  last_state = (_digitalRead(ROT_B) << 1) | _digitalRead(ROT_A);
  interrupts();
}

class I2C {
public:
#if(F_MCU > 20000000)
  #define I2C_DELAY   6
#else
  #define I2C_DELAY   4    // Determines I2C Speed (2=939kb/s (too fast!!); 3=822kb/s;
4=731kb/s; 5=658kb/s; 6=598kb/s). Increase this value when you get I2C tx errors (E05);
```

```
decrease this value when you get a CPU overload (E01). An increment eats ~3.5% CPU
load; minimum value is 3 on my QCX, resulting in 84.5% CPU load
#endif
  #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();
  /*#define SendByte(data) \
    SendBit(data, 1 << 7) \
    SendBit(data, 1 << 6) \
    SendBit(data, 1 << 5) \
    SendBit(data, 1 << 4) \
    SendBit(data, 1 << 3) \
    SendBit(data, 1 << 2) \
    SendBit(data, 1 << 1) \
    SendBit(data, 1 << 0) \
    I2C_SDA_HI();  // recv ACK \
    DELAY(I2C_DELAY);     \
    I2C_SCL_HI();         \
    I2C_SCL_LO();*/
  inline void SendByte(uint8_t data){
    SendBit(data, 1 << 7);
    SendBit(data, 1 << 6);
    SendBit(data, 1 << 5);
    SendBit(data, 1 << 4);
    SendBit(data, 1 << 3);
    SendBit(data, 1 << 2);
    SendBit(data, 1 << 1);
    SendBit(data, 1 << 0);
    I2C_SDA_HI();  // recv ACK
    DELAY(I2C_DELAY);
    I2C_SCL_HI();
```

```
    I2C_SCL_LO();
  }
  inline uint8_t RecvBit(uint8_t mask){
    I2C_SCL_HI();
    uint16_t i = 60000;
    for(;(!I2C_SCL_GET()) && i; i--);  // wait util slave release SCL to HIGH (meaning
data valid), or timeout at 3ms
    //if(!i){ lcd.setCursor(0, 1); lcd.print(F("E07 I2C timeout")); }
    uint8_t data = I2C_SDA_GET();
    I2C_SCL_LO();
    return (data) ? mask : 0;
  }
  inline uint8_t RecvByte(uint8_t last){
    uint8_t data = 0;
    data |= RecvBit(1 << 7);
    data |= RecvBit(1 << 6);
    data |= RecvBit(1 << 5);
    data |= RecvBit(1 << 4);
    data |= RecvBit(1 << 3);
    data |= RecvBit(1 << 2);
    data |= RecvBit(1 << 1);
    data |= RecvBit(1 << 0);
    if(last){
      I2C_SDA_HI();  // NACK
    } else {
      I2C_SDA_LO();  // ACK
    }
    DELAY(I2C_DELAY);
    I2C_SCL_HI();
    I2C_SDA_HI();    // restore SDA for read
    I2C_SCL_LO();
    return data;
  }
  inline void resume(){
  #ifdef LCD_RS_PORTIO
    I2C_PORT &= ~I2C_SDA; // pin sharing SDA/LCD_RS mitigation
  #endif
  }
  inline void suspend(){
    I2C_SDA_LO();         // pin sharing SDA/LCD_RS: pull-down LCD_RS; QCXLiquidCrystal
require this for any operation
  }

  void begin(){};
  void beginTransmission(uint8_t addr){ start(); SendByte(addr << 1);  };
  bool write(uint8_t byte){ SendByte(byte); return 1; };
  uint8_t endTransmission(){ stop(); return 0; };
};

#define log2(n) (log(n) / log(2))

// /*
I2C i2c;
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 uint32_t _mod;
  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")))

  volatile uint32_t fxtal = F_XTAL;
```

```
#define NEW_TX 1
#ifdef NEW_TX
  inline void FAST freq_calc_fast(int16_t df)  // note: relies on cached variables:
_msb128, _msa128min512, _div, _fout, fxtal
  {
    #define _MSC  0x10000
    uint32_t msb128 = _msb128 + ((int64_t)(_div * (int32_t)df) * _MSC * 128) / fxtal;

    uint16_t msp1 = _msa128min512 + msb128 / _MSC; // = 128 * _msa + msb128 / _MSC -
512;
    uint16_t msp2 = msb128; // = msb128 % _MSC;  assuming MSC is covering exact
uint16_t so the mod operation can dissapear (and the upper BB2 byte) // = 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 _MSC !  assuming that BB2(msp2) is always 0 -> so reg is constant
    pll_regs[6] = BB1(msp2);
    pll_regs[7] = BB0(msp2);
  }

  inline void SendPLLRegisterBulk(){
    i2c.start();
    i2c.SendByte(SI5351_ADDR << 1);
    i2c.SendByte(26+0*8 + 4);  // Write to PLLA
    //i2c.SendByte(26+1*8 + 4);  // Write to PLLB
    i2c.SendByte(pll_regs[4]);
    i2c.SendByte(pll_regs[5]);
    i2c.SendByte(pll_regs[6]);
    i2c.SendByte(pll_regs[7]);
    i2c.stop();
  }
#else  // !NEW_TX
  inline void FAST freq_calc_fast(int16_t df)  // note: relies on cached variables:
_msb128, _msa128min512, _div, _fout, fxtal
  {
    #define _MSC  0x80000  //0x80000: 98% CPU load   0xFFFFF: 114% CPU load
    uint32_t msb128 = _msb128 + ((int64_t)(_div * (int32_t)df) * _MSC * 128) / fxtal;
    //uint32_t msb128 = ((int64_t)(_div * (int32_t)df + _mod) * _MSC * 128) / fxtal; //
@pre: 14<=_div<=144, |df|<=5000, _mod<=1800e3 (for fout<30M), _MSC=524288

    //#define _MSC  (F_XTAL/128)   // MSC exact multiple of F_XTAL (and maximized to
fit in max. span 1048575)
    //uint32_t msb128 = (_div * (int32_t)df + _mod);

    //#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 * _MSC);
    //uint32_t msb128 = xmsb * 5*(32/32) - (xmsb/32);  // msb128 = xmsb * 159/32, where
159/32 = 128 * 0xFFFFF / fxtal; fxtal=27e6

    //#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;
    //uint32_t msp1 = _msa128min512;  // = 128 * _msa + msb128 / _MSC - 512;  assuming
msb128 < _MSC, so that msp1 is constant
    //uint32_t msp2 = msb128;  // = msb128 - msb128/_MSC * _MSC, assuming msb128 < _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 _MSC !
    pll_regs[6] = BB1(msp2);
    pll_regs[7] = BB0(msp2);
  }

  inline void SendPLLRegisterBulk(){
    i2c.start();
    i2c.SendByte(SI5351_ADDR << 1);
    i2c.SendByte(26+0*8 + 3);  // Write to PLLA
    //i2c.SendByte(26+1*8 + 3);  // Write to PLLB
    i2c.SendByte(pll_regs[3]);
    i2c.SendByte(pll_regs[4]);
    i2c.SendByte(pll_regs[5]);
    i2c.SendByte(pll_regs[6]);
    i2c.SendByte(pll_regs[7]);
    i2c.stop();
  }
#endif // !NEW_TX

  void SendRegister(uint8_t reg, uint8_t* data, uint8_t n){
    i2c.start();
    i2c.SendByte(SI5351_ADDR << 1);
    i2c.SendByte(reg);
    while (n--) i2c.SendByte(*data++);
    i2c.stop();
  }
  void SendRegister(uint8_t reg, uint8_t val){ SendRegister(reg, &val, 1); }
  int16_t iqmsa; // to detect a need for a PLL reset
///*
  enum ms_t { PLLA=0, PLLB=1, MSNA=-2, MSNB=-1, MS0=0, MS1=1, MS2=2, MS3=3, MS4=4,
MS5=5 };

  void ms(int8_t n, uint32_t div_nom, uint32_t div_denom, uint8_t pll = PLLA, uint8_t
_int = 0, uint16_t phase = 0, uint8_t rdiv = 0){
    uint16_t msa; uint32_t msb, msc, msp1, msp2, msp3;
    msa = div_nom / div_denom;     // integer part: msa must be in range 15..90 for
PLL, 8+1/1048575..900 for MS
    if(msa == 4) _int = 1;  // To satisfy the MSx_INT=1 requirement of AN619, section
4.1.3 which basically says that for MS divider a value of 4 and integer mode must be
used
    msb = (_int) ? 0 : (((uint64_t)(div_nom % div_denom)*_MSC) / div_denom); //
fractional part
    msc = (_int) ? 1 : _MSC;
    //lcd.setCursor(0, 0); lcd.print(n); lcd.print(":"); lcd.print(msa); lcd.print("
"); lcd.print(msb); lcd.print(" "); lcd.print(msc); lcd.print(F("    ")); delay(500);
    msp1 = 128*msa + 128*msb/msc - 512;
    msp2 = 128*msb - 128*msb/msc * msc;
    msp3 = msc;
    uint8_t ms_regs[8] = { BB1(msp3), BB0(msp3), BB2(msp1) | (rdiv<<4) | ((msa ==
4)*0x0C), BB1(msp1), BB0(msp1), BB2(((msp3 & 0x0F0000)<<4) | msp2), BB1(msp2),
BB0(msp2) };
    SendRegister(n*8+42, ms_regs, 8); // Write to MSx
    if(n < 0){
      SendRegister(n+16+8, 0x80|(0x40*_int)); // MSNx PLLn: 0x40=FBA_INT; 0x80=CLKn_PDN
    } else {
      //SendRegister(n+16, ((pll)*0x20)|0x0C|0|(0x40*_int));  // MSx CLKn:
0x0C=PLLA,0x2C=PLLB local msynth; 0=2mA; 0x40=MSx_INT; 0x80=CLKx_PDN
      SendRegister(n+16, ((pll)*0x20)|0x0C|3|(0x40*_int));  // MSx CLKn:
0x0C=PLLA,0x2C=PLLB local msynth; 3=8mA; 0x40=MSx_INT; 0x80=CLKx_PDN
      SendRegister(n+165, (!_int) * phase * msa / 90);      // when using: make sure to
configure MS in fractional-mode, perform reset afterwards
    }
  }

  void phase(int8_t n, uint32_t div_nom, uint32_t div_denom, uint16_t phase){
SendRegister(n+165, phase * (div_nom / div_denom) / 90); }  // when using: make sure to
configure MS in fractional-mode!, perform reset afterwards
```

```
  void reset(){ SendRegister(177, 0xA0); } // 0x20 reset PLLA; 0x80 reset PLLB

  void oe(uint8_t mask){ SendRegister(3, ~mask); } // output-enable mask: CLK2=4;
CLK1=2; CLK0=1

  void freq(int32_t fout, uint16_t i, uint16_t q){  // Set a CLK0,1,2 to fout Hz with
phase i, q (on PLLA)
      uint8_t rdiv = 0; // CLK pin sees fout/(2^rdiv)
      if(fout > 300000000){ i/=3; q/=3; fout/=3; }  // for higher freqs, use 3rd
harmonic
      if(fout < 500000){ rdiv = 7; fout *= 128; } // Divide by 128 for fout 4..500kHz
      uint16_t d; if(fout < 30000000) d = (16 * fxtal) / fout; else d = (32 * fxtal) /
fout;  // Integer part  .. maybe 44?
      if(fout < 3500000) d = (7 * fxtal) / fout;  // PLL at 189MHz to cover 160m
(freq>1.48MHz) when using 27MHz crystal
      if(fout > 140000000) d = 4; // for f=140..300MHz; AN619; 4.1.3, this implies
integer mode
      if(d % 2) d++;  // even numbers preferred for divider (AN619 p.4 and p.6)
      if( (d * (fout - 5000) / fxtal) != (d * (fout + 5000) / fxtal) ) d += 2; // Test
if multiplier remains same for freq deviation +/- 5kHz, if not use different divider to
make same
      uint32_t fvcoa = d * fout;  // Variable PLLA VCO frequency at integer multiple of
fout at around 27MHz*16 = 432MHz
      // si5351 spectral purity considerations: https://groups.io/g/QRPLabs/message
/42662

      ms(MSNA, fvcoa, fxtal);                   // PLLA in fractional mode
      //ms(MSNB, fvcoa, fxtal);
      ms(MS0,  fvcoa, fout, PLLA, 0, i, rdiv);  // Multisynth stage with integer
divider but in frac mode due to phase setting
      ms(MS1,  fvcoa, fout, PLLA, 0, q, rdiv);
      ms(MS2,  fvcoa, fout, PLLA, 0, 0, rdiv);
      if(iqmsa != ((i-q)*((uint16_t)(fvcoa/fout))/90)){ iqmsa = (i-q)*((uint16_t)
(fvcoa/fout))/90; reset(); }
      oe(0b00000011);  // output enable CLK0, CLK1

#ifdef x
      ms(MSNA, fvcoa, fxtal);
      ms(MSNB, fvcoa, fxtal);
      #define F_DEV 4
      ms(MS0,  fvcoa, (fout + F_DEV), PLLA, 0, 0, rdiv);
      ms(MS1,  fvcoa, (fout + F_DEV), PLLA, 0, 0, rdiv);
      ms(MS2,  fvcoa, fout, PLLA, 0, 0, rdiv);
      reset();
      ms(MS0,  fvcoa, fout, PLLA, 0, 0, rdiv);
      delayMicroseconds(F_MCU/16000000 * 1000000UL/F_DEV);  // Td = 1/(4 * Fdev) phase-
shift   https://tj-lab.org/2020/08/27/si5351%e5%8d%98%e4%bd%93%e3%81%a73mhz%e4%bb%a5%e4
%b8%8b%e3%81%ae%e7%9b%b4%e4%ba%a4%e4%bf%a1%e5%8f%b7%e3%82%92%e5%87%ba%e5%8a%9b%e3%81
%99%e3%82%8b/
      ms(MS1,  fvcoa, fout, PLLA, 0, 0, rdiv);
      oe(0b00000011);  // output enable CLK0, CLK1
#endif
      _fout = fout;  // cache
      _div = d;
      _msa128min512 = fvcoa / fxtal * 128 - 512;
      _msb128=((uint64_t)(fvcoa % fxtal)*_MSC*128) / fxtal;
      //_mod = fvcoa % fxtal;
  }

  void freqb(uint32_t fout){  // Set a CLK2 to fout Hz (on PLLB)
      uint16_t d = (16 * fxtal) / fout;
      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 = 432MHz

      ms(MSNB, fvcoa, fxtal);
      ms(MS2,  fvcoa, fout, PLLB, 0, 0, 0);
  }
```

```
  uint8_t RecvRegister(uint8_t reg){
    i2c.start();  // Data write to set the register address
    i2c.SendByte(SI5351_ADDR << 1);
    i2c.SendByte(reg);
    i2c.stop();
    i2c.start(); // Data read to retrieve the data from the set address
    i2c.SendByte((SI5351_ADDR << 1) | 1);
    uint8_t data = i2c.RecvByte(true);
    i2c.stop();
    return data;
  }
  void powerDown(){
    SendRegister(3, 0b11111111); // Disable all CLK outputs
    SendRegister(24, 0b00000000); // Disable state: LOW state when disabled
    SendRegister(25, 0b00000000); // Disable state: LOW state when disabled
    for(int addr = 16; addr != 24; addr++) SendRegister(addr, 0b10000000);  // Conserve
power when output is disabled
    SendRegister(187, 0);        // Disable fanout (power-safe)
    // To initialise things as they should:
    SendRegister(149, 0);        // Disable spread spectrum enable
    SendRegister(183, 0b11010010);  // Internal CL = 10 pF (default)
  }
  #define SI_CLK_OE 3

};
static SI5351 si5351;



enum dsp_cap_t { ANALOG, DSP, SDR };
const uint8_t ssb_cap = 1;
const uint8_t dsp_cap = SDR;
enum mode_t { LSB, USB, CW, FM, AM };
volatile uint8_t mode = USB;
volatile uint16_t numSamples = 0;
volatile uint8_t tx = 0;
volatile uint8_t filt = 0;

inline void _vox(bool trigger)
{
  if(trigger){
    tx = (tx) ? 254 : 255; // hangtime = 255 / 4402 = 58ms (the time that TX at least
stays on when not triggered again). tx == 255 when triggered first, 254 follows for
subsequent triggers, until tx is off.
  } else {
    if(tx) tx--;
  }
}

#define F_SAMP_TX 4810 //4805 // 4402 // (Design) ADC sample-rate; is best a multiple
of _UA and fits exactly in OCR2A = ((F_CPU / 64) / F_SAMP_TX) - 1 , should not exceed
CPU utilization
#if(F_MCU != 20000000)
const int16_t _F_SAMP_TX = (F_MCU * 4810LL / 20000000);  // Actual ADC sample-rate;
used for phase calculations
#else
#define _F_SAMP_TX  F_SAMP_TX
#endif
#define _UA  (_F_SAMP_TX)      //360  // unit angle; integer representation of one full
circle turn or 2pi radials or 360 degrees, should be a integer divider of F_SAMP_TX and
maximized to have higest precision
#define MAX_DP  ((filt == 0) ? _UA : (filt == 3) ? _UA/4 : _UA/2)     //(_UA/2) // the
occupied SSB bandwidth can be further reduced by restricting the maximum phase change
(set MAX_DP to _UA/2).
#define CARRIER_COMPLETELY_OFF_ON_LOW  1    // disable oscillator on low amplitudes, to
prevent potential unwanted biasing/leakage through PA circuit
#define MULTI_ADC  1  // multiple ADC conversions for more sensitive (+12dB) microphone
input
//#define TX_CLK0_CLK1  1   // use CLK0, CLK1 for TX (instead of CLK2), you may enable
and use NTX pin for enabling the TX path (this is like RX pin, except that RX may also
```

```
be used as attenuator)
//#define QUAD  1       // invert TX signal for phase changes > 180

inline int16_t arctan3(int16_t q, int16_t i)  // error ~ 0.8 degree
{ // source: [1] http://www-labs.iro.umontreal.ca/~mignotte/IFT2425/Documents
/EfficientApproximationArctgFunction.pdf
//#define _atan2(z)  (_UA/8  + _UA/22) * z  // very much of a simplification...not
accurate at all, but fast
#define _atan2(z)  (_UA/8 - _UA/22 * z + _UA/22) * z  //derived from (5) [1]
  //#define _atan2(z)  (_UA/8 - _UA/24 * z + _UA/24) * z  //derived from (7) [1]
  int16_t r;
  if(abs(q) > abs(i))
    r = _UA / 4 - _atan2(abs(i) / abs(q));        // arctan(z) = 90-arctan(1/z)
  else
    r = (i == 0) ? 0 : _atan2(abs(q) / abs(i));   // arctan(z)
  r = (i < 0) ? _UA / 2 - r : r;                  // arctan(-z) = -arctan(z)
  return (q < 0) ? -r : r;                        // arctan(-z) = -arctan(z)
}

#define magn(i, q) (abs(i) > abs(q) ? abs(i) + abs(q) / 4 : abs(q) + abs(i) / 4) //
approximation of: magnitude = sqrt(i*i + q*q); error 0.95dB

uint8_t lut[256];
volatile uint8_t amp;
volatile uint8_t vox_thresh = (1 << 0); //(1 << 2);
volatile uint8_t drive = 2;   // hmm.. drive>2 impacts cpu load..why?

volatile uint8_t quad = 0;

inline int16_t ssb(int16_t in)
{
  static int16_t dc, z1;

  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;       // fast moving average
  dc = (in + dc) / 2;        // average
  int16_t ac = (in - dc);   // DC decoupling
  //v[15] = ac;// - z1;        // high-pass (emphasis) filter
  v[15] = (ac + z1) / 2;           // low-pass filter with notch at Fs/2
  z1 = ac;

  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 rejection in
400..1900Hz (@4kSPS) when used in image-rejection scenario; (Hilbert transform require
5 additional bits)

  uint16_t _amp = magn(i, q);
#ifdef CARRIER_COMPLETELY_OFF_ON_LOW
  _vox(_amp > vox_thresh);
#else
  if(vox) _vox(_amp > vox_thresh);
#endif
  //_amp = (_amp > vox_thresh) ? _amp : 0;   // vox_thresh = 4 is a good setting
  //if(!(_amp > vox_thresh)) return 0;

  _amp = _amp << (drive);
  _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 QUAD
  if(dp >= (_UA/2)){ dp = dp - _UA/2; quad = !quad; }
#endif

#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);
}

#define MIC_ATTEN  0  // 0*6dB attenuation (note that the LSB bits are quite noisy)
volatile int8_t mox = 0;
volatile int8_t volume = 11;

// 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 over I2C.
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-delays in key-
shaping circuit filter can settle)
  si5351.SendPLLRegisterBulk();       // submit frequency registers to SI5351 over
731kbit/s I2C (transfer takes 64/731 = 88us, then PLL-loopfilter probably needs 50us to
stabalize)
#ifdef QUAD
#ifdef TX_CLK0_CLK1
  si5351.SendRegister(16, (quad) ? 0x1f : 0x0f);  // Invert/non-invert CLK0 in case of
a huge phase-change
  si5351.SendRegister(17, (quad) ? 0x1f : 0x0f);  // Invert/non-invert CLK1 in case of
a huge phase-change
#else
  si5351.SendRegister(18, (quad) ? 0x1f : 0x0f);  // Invert/non-invert CLK2 in case of
a huge phase-change
#endif
#endif //QUAD
  OCR1BL = amp;                       // submit amplitude to PWM register (takes about
1/32125 = 31us+/-31us to propagate) -> amplitude-phase-alignment error is about 30-50us
  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);
#define AF_BIAS   32
```

```
  _adc = (adc/4 - (512 - AF_BIAS));        // now make sure that we keep a postive bias
offset (to prevent the phase swapping 180 degrees and potentially causing negative
feedback (RFI)
#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-delays in key-
shaping circuit filter can settle)
  si5351.SendPLLRegisterBulk();       // submit frequency registers to SI5351 over
731kbit/s I2C (transfer takes 64/731 = 88us, then PLL-loopfilter probably needs 50us to
stabalize)
  OCR1BL = amp;                          // submit amplitude to PWM register (takes about
1/32125 = 31us+/-31us to propagate) -> amplitude-phase-alignment error is about 30-50us
  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(tx == 1){ OCR1BL = 0; si5351.SendRegister(SI_CLK_OE, 0b11111111); }   // disable
carrier
#ifdef TX_CLK0_CLK1
  if(tx == 255){ si5351.SendRegister(SI_CLK_OE, 0b11111100); } // enable carrier
#else //TX_CLK2
  if(tx == 255){ si5351.SendRegister(SI_CLK_OE, 0b11111011); } // enable carrier
#endif //TX_CLK0_CLK1
#endif

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

volatile uint16_t acc;
volatile uint32_t cw_offset;
volatile uint8_t cw_tone = 1;
const uint32_t tones[] = { F_MCU * 700ULL / 20000000, F_MCU * 600ULL / 20000000, F_MCU
* 700ULL / 20000000};

volatile int16_t p_sin = 0;   // initialized with A*sin(0) = 0
volatile int16_t n_cos = 448/2; // initialized with A*cos(t) = A
inline void process_minsky() // Minsky circle sample [source: https://www.cl.cam.ac.uk
/~am21/hakmemc.html, ITEM 149]: p_sin+=n_cos*2*PI*f/fs; n_cos-=p_sin*2*PI*f/fs;
{
  uint8_t alpha100 = tones[cw_tone]/*cw_offset*/ * 628 / _F_SAMP_TX;  // alpha = f_tone
* 2 * pi / fs
  p_sin += alpha100 * n_cos / 100;
  n_cos -= alpha100 * p_sin / 100;
}

// CW Key-click shaping, ramping up/down amplitude with sample-interval of 60us. Tnx:
Yves HB9EWY https://groups.io/g/ucx/message/5107
const uint8_t ramp[] PROGMEM = { 255, 254, 252, 249, 245, 239, 233, 226, 217, 208, 198,
187, 176, 164, 152, 139, 127, 115, 102, 90, 78, 67, 56, 46, 37, 28, 21, 15, 9, 5, 2 };
// raised-cosine(i) = 255 * sq(cos(HALF_PI * i/32))

void dummy()
{
}

void dsp_tx_cw()
{ // jitter dependent things first
#ifdef KEY_CLICK
  if(OCR1BL < lut[255]) { //check if already ramped up: ramp up of amplitude
     for(uint16_t i = 31; i != 0; i--) {   // soft rising slope against key-clicks
```

```
        OCR1BL = lut[pgm_read_byte_near(ramp[i])];
        delayMicroseconds(60);
      }
    }
#endif // KEY_CLICK
  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 advance (about 140us) of phase-change, so that phase-delays in key-
shaping circuit filter can settle)
  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];
}

void dsp_tx_fm()
{ // jitter dependent things first
  ADCSRA |= (1 << ADSC);    // start next ADC conversion (trigger ADC interrupt if ADIE
flag is set)
  OCR1BL = lut[255];                     // submit amplitude to PWM register (actually
this is done in advance (about 140us) of phase-change, so that phase-delays in key-
shaping circuit filter can settle)
  si5351.SendPLLRegisterBulk();       // submit frequency registers to SI5351 over
731kbit/s I2C (transfer takes 64/731 = 88us, then PLL-loopfilter probably needs 50us to
stabalize)
  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);
  int16_t df = in;
  si5351.freq_calc_fast(df);           // calculate SI5351 registers based on frequency
shift and carrier frequency
}

#define EA(y, x, one_over_alpha)  (y) = (y) + ((x) - (y)) / (one_over_alpha); //
exponental averaging [Lyons 13.33.1]
#define MLEA(y, x, L, M)  (y)  = (y) + ((((x) - (y)) >> (L)) - (((x) - (y)) >> (M)));
// multiplierless exponental averaging [Lyons 13.33.1], with alpha=1/2^L - 1/2^M



const char m2c[] PROGMEM = "~
ETIANMSURWDKGOHVF*L*PJBXCYZQ**54S3***2**+***J16=/***H*7*G*8*90***********?_****\"**.**
**@***'**-********;!*)*****,****:****";

#ifdef CW_MESSAGE
#define MENU_STR  1

uint8_t delayWithKeySense(uint32_t ms){
  uint32_t event = millis() + ms;
  for(; millis() < event;){
    wdt_reset();
    if(digitalRead(BUTTONS) || !digitalRead(DAH) || !digitalRead(DIT)){
      for(; digitalRead(BUTTONS);) wdt_reset();  // wait until buttons released
      return 1;  // stop when button/key pressed
```

```
    }
  }
  return 0;
}
#ifdef CW_MESSAGE_EXT
char cw_msg[6][48] = { "CQ CQ CQ DE EA2EHC EA2EHC PSE K", "UR RST 599 5NN", "NAME IS
PABLO PABLO HW?", "QTH SANTANDER SANTANDER", "73 TU E E", "EA2EHC" };
#else
char cw_msg[1][48] = { "CQ CQ CQ DE EA2EHC EA2EHC PSK K" };
#endif
uint8_t cw_msg_interval = 5; // number of seconds CW message is repeated
uint32_t cw_msg_event = 0;
uint8_t cw_msg_id = 0; // selected message

int cw_tx(char ch){    // Transmit message in CW
  char sym;
  for(uint8_t j = 0; (sym = pgm_read_byte_near(m2c + j)); j++){  // lookup msg[i] in
m2c, skip if not found
    if(sym == ch){  // found -> transmit CW character j
      wdt_reset();
      uint8_t k = 0x80; for(; !(j & k); k >>= 1); k >>= 1; // shift start of cw code to
MSB
      if(k == 0) delay(ditTime * 4); // space -> add word space
      else {
        for(; k; k >>= 1){ // send dit/dah one by one, until everythng is sent
          switch_rxtx(1);  // key-on  tx
          if(delayWithKeySense(ditTime * ((j & k) ? 3 : 1))){ switch_rxtx(0); return 1;
} // symbol: dah or dih length
          switch_rxtx(0);  // key-off tx
          if(delayWithKeySense(ditTime)) return 1;   // add symbol space
        }
        if(delayWithKeySense(ditTime * 2)) return 1; // add letter space
      }
      break; // next character
    }
  }
  return 0;
}

int cw_tx(char* msg){
  for(uint8_t i = 0; msg[i]; i++){  // loop over message
    lcd.setCursor(0, 0); lcd.print(i); lcd.print("    ");
    if(cw_tx(msg[i])) return 1;
  }
  return 0;
}
#endif // CW_MESSAGE

volatile uint8_t menumode = 0;  // 0=not in menu, 1=selects menu item, 2=selects
parameter value

#ifdef CW_DECODER
volatile uint8_t cwdec = 1;
static int32_t avg = 256;
static uint8_t sym;
static uint32_t amp32 = 0;
volatile uint32_t _amp32 = 0;
static char out[] = "                ";
volatile uint8_t cw_event = false;

void printsym(){
  if(sym<128){ char ch=pgm_read_byte_near(m2c + sym); if(ch != '*'){ for(int i=0;
i!=15;i++) out[i]=out[i+1];
  out[15] = ch;
#ifdef CW_LEARN
    letra = ch;
#endif
  cw_event = true; }  }   // update LCD
  sym=1;
}
```

```
int realstate = LOW;
int realstatebefore = LOW;
int filteredstate = LOW;
int filteredstatebefore = LOW;
int nbtime = 16;  // 6 // ms noise blanker
long starttimehigh;
long highduration;
long hightimesavg;
long lowtimesavg;
long startttimelow;
long lowduration;
long laststarttime = 0;
int wpm = 25;

inline void cw_decode()
{
  int32_t in = _amp32;
  EA(avg, in, (1 << 8));
  realstate = (in>(avg*1/2));  // threshold

  // here we clean up the state with a noise blanker
  if(realstate != realstatebefore){
    laststarttime = millis();
  }
  if((millis() - laststarttime) > nbtime){
    if(realstate != filteredstate){
      filteredstate = realstate;
      //dec2();
    }
  } else avg += avg/100; // keep threshold above noise spikes (increase threshold with
1%)

  dec2();
  realstatebefore = realstate;
}

//#define NEW_CW  1   // CW decoder portions from by Hjalmar Skovholm Hansen OZ1JHM,
source: http://www.skovholm.com/decoder11.ino
#ifdef NEW_CW
void dec2()
{
 // Then we do want to have some durations on high and low
 if(filteredstate != filteredstatebefore){
 if(menumode == 0){ lcd.noCursor(); lcd.setCursor(15, 1); lcd.print((filteredstate) ?
'R' : ' '); stepsize_showcursor(); }

  if(filteredstate == HIGH){
    starttimehigh = millis();
    lowduration = (millis() - startttimelow);
  }

  if(filteredstate == LOW){
    startttimelow = millis();
    highduration = (millis() - starttimehigh);
    if(highduration < (2*hightimesavg) || hightimesavg == 0){
      hightimesavg = (highduration+hightimesavg+hightimesavg)/3;     // now we know avg
dit time ( rolling 3 avg)
    }
    if(highduration > (5*hightimesavg)){
      hightimesavg = highduration/3;     // if speed decrease fast ..
      //hightimesavg = highduration+hightimesavg;     // if speed decrease fast ..
    }
  }
 }

 // now we will check which kind of baud we have - dit or dah, and what kind of pause
we do have 1 - 3 or 7 pause, we think that hightimeavg = 1 bit
 if(filteredstate != filteredstatebefore){
  if(filteredstate == LOW){  //// we did end a HIGH
```

```
    if(highduration < (hightimesavg*2) && highduration > (hightimesavg*0.6)){ /// 0.6
filter out false dits
      sym=(sym<<1)|(0);        // insert dit (0)
    }
    if(highduration > (hightimesavg*2) && highduration < (hightimesavg*6)){
      sym=(sym<<1)|(1);        // insert dah (1)
      wpm = (wpm + (1200/((highduration)/3) * 4/3))/2;
    }
  }

   if(filteredstate == HIGH){  // we did end a LOW
     float lacktime = 1;
     if(wpm > 25)lacktime=1.0; // when high speeds we have to have a little more pause
before new letter or new word
     if(wpm > 30)lacktime=1.2;
     if(wpm > 35)lacktime=1.5;

     if(lowduration > (hightimesavg*(1.75*lacktime)) && lowduration < hightimesavg*
(5*lacktime)){ // letter space
     //if(lowduration > (hightimesavg*(2*lacktime)) && lowduration < hightimesavg*
(5*lacktime)){ // letter space
       printsym();
   }
   if(lowduration >= hightimesavg*(5*lacktime)){ // word space
     printsym();
     printsym();  // print space
   }
  }
 }

 // write if no more letters
  if((millis() - startttimelow) > (highduration * 6) && (sym > 1)){
    printsym();
  }

  filteredstatebefore = filteredstate;
}

#else // OLD_CW

void dec2()
{
 if(filteredstate != filteredstatebefore){ // then we do want to have some durations on
high and low
  if(menumode == 0){ lcd.noCursor(); lcd.setCursor(15, 1); lcd.print((filteredstate) ?
'R' : ' '); stepsize_showcursor(); }

  if(filteredstate == HIGH){
    starttimehigh = millis();
    lowduration = (millis() - startttimelow);
    //highduration = 0;

    if((sym > 1) && lowduration > (hightimesavg*2)/* && lowduration < hightimesavg*
(5*lacktime)*/){ // letter space
      printsym();
      wpm = (1200/hightimesavg * 4/3);
      //if(lowduration >= hightimesavg*(5)){ sym=1; printsym(); } // (print additional
space) word space
    }
    if(lowduration >= hightimesavg*(5)){ sym=1; printsym(); } // (print additional
space) word space
  }

  if(filteredstate == LOW){
    startttimelow = millis();
    highduration = (millis() - starttimehigh);
    //lowduration = 0;
    if(highduration < (2*hightimesavg) || hightimesavg == 0){
      hightimesavg = (highduration+hightimesavg+hightimesavg)/3;     // now we know avg
dit time (rolling 3 avg)
```

```
    }
    if(highduration > (5*hightimesavg)){
      hightimesavg = highduration/3;     // if speed decrease fast ..
      //hightimesavg = highduration+hightimesavg;     // if speed decrease fast ..
    }
    if(highduration > (hightimesavg/2)) sym=(sym<<1)|(highduration > (hightimesavg*2));
// dit (0) or dash (1)
  }
 }

 if(((millis() - startttimelow) > hightimesavg*(6)) && (sym > 1)){
 //if(((millis() - startttimelow) > hightimesavg*(12)) && (sym > 1)){
   //if(sym == 2) sym = 1; else // skip E E E E E
   printsym();  // write if no more letters
   //sym=0; printsym(); // print special char
   //startttimelow = millis();
 }

 filteredstatebefore = filteredstate;
}
#endif //OLD_CW
#endif  //CW_DECODER

#define F_SAMP_PWM (78125/1)
//#define F_SAMP_RX 78125  // overrun, do not use
#define F_SAMP_RX 62500
//#define F_SAMP_RX 52083
//#define F_SAMP_RX 44643
//#define F_SAMP_RX 39062
//#define F_SAMP_RX 34722
//#define F_SAMP_RX 31250
//#define F_SAMP_RX 28409
#define F_ADC_CONV (192307/2)  //was 192307/1, but as noted this produces clicks in
audio stream. Slower ADC clock cures this (but is a problem for VOX when sampling mic-
input simulatanously).

#ifdef FAST_AGC
volatile uint8_t agc = 2;
#else
volatile uint8_t agc = 1;
#endif
volatile uint8_t nr = 0;
volatile uint8_t att = 0;
volatile uint8_t att2 = 2;  // Minimum att2 increased, to prevent numeric overflow on
strong signals
volatile uint8_t _init = 0;

// Old AGC algorithm which only increases gain, but does not decrease it for very
strong signals.
// Maximum possible gain is x32 (in practice, x31) so AGC range is x1 to x31 = 30dB
approx.
// Decay time is fine (about 1s) but attack time is much slower than I like.
// For weak/medium signals it aims to keep the sample value between 1024 and 2048.
static int16_t gain = 1024;
inline int16_t process_agc_fast(int16_t in)
{
  int16_t out = (gain >= 1024) ? (gain >> 10) * in : in;
  int16_t accum = (1 - abs(out >> 10));
  if((INT16_MAX - gain) > accum) gain = gain + accum;
  if(gain < 1) gain = 1;
  return out;
}

// Contribution by Alan, M0PUB: Experimental new AGC algorithm.
// ASSUMES: Input sample values are constrained to a maximum of +/-4096 to avoid
integer overflow in earlier
// calculations.
//
// This algorithm aims to keep signals between a peak sample value of 1024 - 1536, with
fast attack but slow
```

```
// decay.
//
// The variable centiGain actually represents the applied gain x 128 - i.e. the numeric
gain applied is centiGain/128
//
// Since the largest valid input sample has a value of +/- 4096, centiGain should never
be less than 32 (i.e.
// a 'gain' of 0.25). The maximum value for centiGain is 32767, and hence a gain of
255. So the AGC range
// is 0.25:255, or approx. 60dB.
//
// Variable 'slowdown' allows the decay time to be slowed down so that it is not
directly related to the value
// of centiCount.

static int16_t centiGain = 128;
#define DECAY_FACTOR 400      // AGC decay occurs <DECAY_FACTOR> slower than attack.
static uint16_t decayCount = DECAY_FACTOR;
#define HI(x)  ((x) >> 8)
#define LO(x)  ((x) & 0xFF)

inline int16_t process_agc(int16_t in)
{
  static bool small = true;
  int16_t out;

  if(centiGain >= 128)
    out = (centiGain >> 5) * in;         // net gain >= 1
  else
    out = (centiGain >> 2) * (in >> 3);  // net gain < 1
  out >>= 2;

  if(HI(abs(out)) > HI(1536)){
    centiGain -= (centiGain >> 4);       // Fast attack time when big signal
encountered (relies on CentiGain >= 16)
  } else {
    if(HI(abs(out)) > HI(1024))
      small = false;
    if(--decayCount == 0){               // But slow ramp up of gain when signal
disappears
      if(small){                         // 400 samples below lower threshold -
increase gain
        if(centiGain < (INT16_MAX-(INT16_MAX >> 4)))
          centiGain += (centiGain >> 4);
        else
          centiGain = INT16_MAX;
      }
      decayCount = DECAY_FACTOR;
      small = true;
    }
  }
  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 (switchoff agc
and tune-up volume until noise dissapears, todo:extra volume control needed)
  ac = ac << 3;
  return ac;
}

inline int16_t process_nr_old2(int16_t ac)
{
  int16_t x = ac;
  static int16_t ea1;
  //ea1 = MLEA(ea1, ac, 5, 6); // alpha=0.0156
  ea1 = EA(ea1, ac, 64); // alpha=1/64=0.0156
  //static int16_t ea2;
  //ea2 = EA(ea2, ea1, 64); // alpha=1/64=0.0156
```

```
  return ea1;
}

inline int16_t process_nr(int16_t in)
{
/*
  static int16_t avg;
  avg = EA(avg, abs(in), 64); // alpha=1/64=0.0156
param_c = avg;
*/

/*
  int32_t _avg = 64 * avg;
//  if(_avg > 4) _avg = 4;  // clip
//  uint16_t brs_avgsq = 1 << (_avg * _avg);
  if(_avg > 14) _avg = 14;  // clip
  uint16_t brs_avgsq = 1 << (_avg);


  int16_t inv_gain;
  if(brs_avgsq > 1) inv_gain = brs_avgsq / (brs_avgsq - 1);  // = 1 / (1 - 1/(1 <<
(1*avg*avg)) );
  else inv_gain = 32768;*/

  static int16_t ea1;
  ea1 = EA(ea1, in, 1 << (nr-1) );
  //static int16_t ea2;
  //ea2 = EA(ea2, ea1, inv_gain);

  return ea1;
}
/*
inline int16_t process_nr(int16_t in)
{
  // Exponential moving average and variance (Lyons 13.36.2)
  param_b = EA(param_b, in, 1 << 4);  // avg
  param_c = EA(param_c, (in - param_b) * (in - param_b), 1 << 4);  // variance
}
*/

#define N_FILT 7
//volatile uint8_t filt = 0;
uint8_t prev_filt[] = { 0 , 4 }; // default filter for modes resp. CW, SSB

/* basicdsp filter simulation:
  samplerate=7812
  za0=in
  p1=slider1*10
  p2=slider2*10
  p3=slider3*10
  p4=slider4*10
  zb0=(za0+2*za1+za2)/2-(p1*zb1+p2*zb2)/16
  zc0=(zb0+2*zb1+zb2)/4-(p3*zc1+p4*zc2)/16
  zc2=zc1
  zc1=zc0
  zb2=zb1
  zb1=zb0
  za2=za1
  za1=za0
  out=zc0

  samplerate=7812
  za0=in
  p1=slider1*100+100
  p2=slider2*100
  p3=slider3*100+100
  p4=slider4*100
  zb0=(za0+2*za1+za2)-(-p1*zb1+p2*zb2)/64
  zc0=(zb0-2*zb1+zb2)/8-(-p3*zc1+p4*zc2)/64
```

```
  zc2=zc1
  zc1=zc0
  zb2=zb1
  zb1=zb0
  za2=za1
  za1=za0
  out=zc0/8
*/
inline int16_t filt_var(int16_t za0)  //filters build with www.micromodeler.com
{
  static int16_t za1,za2;
  static int16_t zb0,zb1,zb2;
  static int16_t zc0,zc1,zc2;

  if(filt < 4)
  {  // for SSB filters
    // 1st Order (SR=8kHz) IIR in Direct Form I, 8x8:16
    // M0PUB: There was a bug here, since za1 == zz1 at this point in the code, and the
old algorithm for the 300Hz high-pass was:
    //    za0=(29*(za0-zz1)+50*za1)/64;
    //    zz2=zz1;
    //    zz1=za0;
    // After correction, this filter still introduced almost 6dB attenuation, so I
adjusted the coefficients
    static int16_t zz1,zz2;
    //za0=(29*(za0-zz1)+50*za1)/64;                                  //300-Hz
    zz2=zz1;
    zz1=za0;
    za0=(30*(za0-zz2)+25*za1)/32;                                  //300-Hz

    // 4th Order (SR=8kHz) IIR in Direct Form I, 8x8:16
    switch(filt){
      case 1: zb0=(za0+2*za1+za2)/2-(13*zb1+11*zb2)/16; break;   // 0-2900Hz filter,
first biquad section
      case 2: zb0=(za0+2*za1+za2)/2-(2*zb1+8*zb2)/16; break;     // 0-2400Hz filter,
first biquad section
      //case 3: zb0=(za0+2*za1+za2)/2-(4*zb1+2*zb2)/16; break;     // 0-2400Hz filter,
first biquad section
      case 3: zb0=(za0+2*za1+za2)/2-(0*zb1+4*zb2)/16; break;     //0-1800Hz  elliptic
      //case 3: zb0=(za0+7*za1+za2)/16-(-24*zb1+9*zb2)/16; break;  //0-1700Hz  elliptic
with slope
    }

    switch(filt){
      case 1: zc0=(zb0+2*zb1+zb2)/2-(18*zc1+11*zc2)/16; break;     // 0-2900Hz filter,
second biquad section
      case 2: zc0=(zb0+2*zb1+zb2)/4-(4*zc1+8*zc2)/16; break;       // 0-2400Hz filter,
second biquad section
      //case 3: zc0=(zb0+2*zb1+zb2)/4-(1*zc1+9*zc2)/16; break;       // 0-2400Hz
filter, second biquad section
      case 3: zc0=(zb0+2*zb1+zb2)/4-(0*zc1+4*zc2)/16; break;       //0-1800Hz  elliptic
      //case 3: zc0=(zb0+zb1+zb2)/16-(-22*zc1+47*zc2)/64; break;   //0-1700Hz  elliptic
with slope
    }
   /*switch(filt){
      case 1: zb0=za0; break; //0-4000Hz (pass-through)
      case 2: zb0=(10*(za0+2*za1+za2)+16*zb1-17*zb2)/32; break;    //0-2500Hz  elliptic
-60dB@3kHz
      case 3: zb0=(7*(za0+2*za1+za2)+48*zb1-18*zb2)/32; break;     //0-1700Hz  elliptic
    }

    switch(filt){
      case 1: zc0=zb0; break; //0-4000Hz (pass-through)
      case 2: zc0=(8*(zb0+zb2)+13*zb1-43*zc1-52*zc2)/64; break;   //0-2500Hz  elliptic
-60dB@3kHz
      case 3: zc0=(4*(zb0+zb1+zb2)+22*zc1-47*zc2)/64; break;   //0-1700Hz  elliptic
    }*/

    zc2=zc1;
    zc1=zc0;
```

```
    zb2=zb1;
    zb1=zb0;

    za2=za1;
    za1=za0;

    return zc0;
  } else { // for CW filters
    //   (2nd Order (SR=4465Hz) IIR in Direct Form I, 8x8:16), adding 64x front-gain
(to deal with later division)

#ifdef FILTER_700HZ
    if(cw_tone == 0){
      switch(filt){
        case 4: zb0=(za0+2*za1+za2)/2+(41L*zb1-23L*zb2)/32; break;   //500-1000Hz
        case 5: zb0=5*(za0-2*za1+za2)+(105L*zb1-58L*zb2)/64; break;   //650-840Hz
        case 6: zb0=3*(za0-2*za1+za2)+(108L*zb1-61L*zb2)/64; break;   //650-750Hz
        case 7: zb0=(2*za0-3*za1+2*za2)+(111L*zb1-62L*zb2)/64; break; //630-680Hz
        //case 4: zb0=(0*za0+1*za1+0*za2)+(28*zb1-14*zb2)/16; break; //600Hz+-250Hz
        //case 5: zb0=(0*za0+1*za1+0*za2)+(28*zb1-15*zb2)/16; break; //600Hz+-100Hz
        //case 6: zb0=(0*za0+1*za1+0*za2)+(27*zb1-15*zb2)/16; break; //600Hz+-50Hz
        //case 7: zb0=(0*za0+1*za1+0*za2)+(27*zb1-15*zb2)/16; break; //630Hz+-18Hz
      }

      switch(filt){
        case 4: zc0=(zb0-2*zb1+zb2)/4+(105L*zc1-52L*zc2)/64; break;      //500-1000Hz
        case 5: zc0=((zb0+2*zb1+zb2)+97L*zc1-57L*zc2)/64; break;      //650-840Hz
        case 6: zc0=((zb0+zb1+zb2)+104L*zc1-60L*zc2)/64; break;       //650-750Hz
        case 7: zc0=((zb1)+109L*zc1-62L*zc2)/64; break;               //630-680Hz
        //case 4: zc0=(zb0-2*zb1+zb2)/1+(24*zc1-13*zc2)/16; break; //600Hz+-250Hz
        //case 5: zc0=(zb0-2*zb1+zb2)/4+(26*zc1-14*zc2)/16; break; //600Hz+-100Hz
        //case 6: zc0=(zb0-2*zb1+zb2)/16+(28*zc1-15*zc2)/16; break; //600Hz+-50Hz
        //case 7: zc0=(zb0-2*zb1+zb2)/32+(27*zc1-15*zc2)/16; break; //630Hz+-18Hz
      }
    }
    if(cw_tone == 1)
#endif
    {
      switch(filt){
        //case 4: zb0=(1*za0+2*za1+1*za2)+(90L*zb1-38L*zb2)/64; break; //600Hz+-250Hz
        //case 5: zb0=(1*za0+2*za1+1*za2)/2+(102L*zb1-52L*zb2)/64; break; //600Hz+-
100Hz
        //case 6: zb0=(1*za0+2*za1+1*za2)/2+(107L*zb1-57L*zb2)/64; break; //600Hz+-50Hz
        //case 7: zb0=(0*za0+1*za1+0*za2)+(110L*zb1-61L*zb2)/64; break; //600Hz+-25Hz

        case 4: zb0=(0*za0+1*za1+0*za2)+(114L*zb1-57L*zb2)/64; break; //600Hz+-250Hz
        case 5: zb0=(0*za0+1*za1+0*za2)+(113L*zb1-60L*zb2)/64; break; //600Hz+-100Hz
        case 6: zb0=(0*za0+1*za1+0*za2)+(110L*zb1-62L*zb2)/64; break; //600Hz+-50Hz
        case 7: zb0=(0*za0+1*za1+0*za2)+(110L*zb1-61L*zb2)/64; break; //600Hz+-18Hz
        //case 8: zb0=(0*za0+1*za1+0*za2)+(110L*zb1-60L*zb2)/64; break; //591Hz+-12Hz

        /*case 4: zb0=(0*za0+1*za1+0*za2)+2*zb1-zb2+(-14L*zb1+7L*zb2)/64; break;
//600Hz+-250Hz
        case 5: zb0=(0*za0+1*za1+0*za2)+2*zb1-zb2+(-15L*zb1+4L*zb2)/64; break;
//600Hz+-100Hz
        case 6: zb0=(0*za0+1*za1+0*za2)+2*zb1-zb2+(-14L*zb1+2L*zb2)/64; break;
//600Hz+-50Hz
        case 7: zb0=(0*za0+1*za1+0*za2)+2*zb1-zb2+(-14L*zb1+3L*zb2)/64; break;
//600Hz+-18Hz*/
      }

      switch(filt){
        //case 4: zc0=(zb0-2*zb1+zb2)/4+(95L*zc1-44L*zc2)/64; break; //600Hz+-250Hz
        //case 5: zc0=(zb0-2*zb1+zb2)/8+(104L*zc1-53L*zc2)/64; break; //600Hz+-100Hz
        //case 6: zc0=(zb0-2*zb1+zb2)/16+(106L*zc1-56L*zc2)/64; break; //600Hz+-50Hz
        //case 7: zc0=(zb0-2*zb1+zb2)/32+(112L*zc1-62L*zc2)/64; break; //600Hz+-25Hz

        case 4: zc0=(zb0-2*zb1+zb2)/1+(95L*zc1-52L*zc2)/64; break; //600Hz+-250Hz
        case 5: zc0=(zb0-2*zb1+zb2)/4+(106L*zc1-59L*zc2)/64; break; //600Hz+-100Hz
```

```
        case 6: zc0=(zb0-2*zb1+zb2)/16+(113L*zc1-62L*zc2)/64; break; //600Hz+-50Hz
        case 7: zc0=(zb0-2*zb1+zb2)/32+(112L*zc1-62L*zc2)/64; break; //600Hz+-18Hz
        //case 8: zc0=(zb0-2*zb1+zb2)/64+(113L*zc1-63L*zc2)/64; break; //591Hz+-12Hz

        /*case 4: zc0=(zb0-2*zb1+zb2)/1+zc1-zc2+(31L*zc1+12L*zc2)/64; break; //600Hz+-
250Hz
        case 5: zc0=(zb0-2*zb1+zb2)/4+2*zc1-zc2+(-22L*zc1+5L*zc2)/64; break; //600Hz+-
100Hz
        case 6: zc0=(zb0-2*zb1+zb2)/16+2*zc1-zc2+(-15L*zc1+2L*zc2)/64; break;
//600Hz+-50Hz
        case 7: zc0=(zb0-2*zb1+zb2)/16+2*zc1-zc2+(-16L*zc1+2L*zc2)/64; break; //600Hz+-
18Hz*/
      }
    }
    zc2=zc1;
    zc1=zc0;

    zb2=zb1;
    zb1=zb0;

    za2=za1;
    za1=za0;

    //return zc0 / 64; // compensate the 64x front-end gain
    return zc0 / 8; // compensate the front-end gain
  }
}

inline int16_t _arctan3(int16_t q, int16_t i)
{
  #define __atan2(z)  (_UA/8  + _UA/22) * z  // very much of a simplification...not
accurate at all, but fast
  int16_t r;
  if(abs(q) > abs(i))
    r = _UA / 4 - __atan2(abs(i) / abs(q));        // arctan(z) = 90-arctan(1/z)
  else
    r = (i == 0) ? 0 : _atan2(abs(q) / abs(i));   // arctan(z)
  r = (i < 0) ? _UA / 2 - r : r;                  // arctan(-z) = -arctan(z)
  return (q < 0) ? -r : r;                        // arctan(-z) = -arctan(z)
}

static uint32_t absavg256 = 0;
volatile uint32_t _absavg256 = 0;
volatile int16_t i, q;

inline int16_t slow_dsp(int16_t ac)
{
  static uint8_t absavg256cnt;
  if(!(absavg256cnt--)){ _absavg256 = absavg256; absavg256 = 0; } else absavg256 +=
abs(ac);

  if(mode == AM) {
    ac = magn(i, q);
    //static int16_t last_sample = 1;
    //MLEA(last_sample,ac, 1, 2); // 1/2: alpha = 0.25, Lyons 13.33.2 p.763
    /*{ static int16_t dc;
      dc += (ac - dc) / 2;
      ac = ac - dc; }  // DC decoupling*/
  } else if(mode == FM){
    static int16_t z1;
    int16_t z0 = _arctan3(q, i);
    ac = z0 - z1; // Differentiator
    z1 = z0;
    //ac = ac * (F_SAMP_RX/R) / _UA;  // =ac*3.5 -> skip
  }  // needs: p.12 https://www.veron.nl/wp-content/uploads/2014/01/FmDemodulator.pdf
  else { ; }  // USB, LSB, CW

#ifdef FAST_AGC
  if(agc == 2) {
    ac = process_agc(ac);
```

```
    ac = ac >> (16-volume);
  } else if(agc == 1){
    ac = process_agc_fast(ac);
    ac = ac >> (16-volume);
#else
  if(agc == 1){
    ac = process_agc_fast(ac);
    ac = ac >> (16-volume);
#endif //!FAST_AGC
  } else {
    //ac = ac >> (16-volume);
    if(volume <= 13)    // if no AGC allow volume control to boost weak signals
      ac = ac >> (13-volume);
    else
      ac = ac << (volume-13);
  }
  if(nr) ac = process_nr(ac);

//  if(filt) ac = filt_var(ac) << 2;
  if(filt) ac = filt_var(ac);
/*
  if(mode == CW){
    if(cwdec){  // CW decoder enabled?
      char ch = cw(ac >> 0);
      if(ch){
        for(int i=0; i!=15;i++) out[i]=out[i+1];
        out[15] = ch;
        cw_event = true;
      }
    }
  }*/
#ifdef CW_DECODER
  if(!(absavg256cnt % 64)){ _amp32 = amp32; amp32 = 0; } else amp32 += abs(ac);
#endif  //CW_DECODER
  //if(!(absavg256cnt--)){ _absavg256 = absavg256; absavg256 = 0; } else absavg256 +=
abs(ac);  //hack

  //static int16_t dc;
  //dc += (ac - dc) / 2;
  //dc = (15*dc + ac)/16;
  //dc = (15*dc + (ac - dc))/16;
  //ac = ac - dc;    // DC decoupling

  ac = min(max(ac, -512), 511);
  //ac = min(max(ac, -128), 127);

  return ac;

}



volatile uint8_t cat_streaming = 0;
volatile uint8_t _cat_streaming = 0;

typedef void (*func_t)(void);
volatile func_t func_ptr;
#undef  R  // Decimating 2nd Order CIC filter
#define R 4  // Rate change from 62500/2 kSPS to 7812.5SPS, providing 12dB gain

//#define SIMPLE_RX  1
#ifndef SIMPLE_RX
volatile uint8_t admux[3];
volatile int16_t ocomb, qh;
volatile uint8_t rx_state = 0;

#pragma GCC push_options
#pragma GCC optimize ("Ofast")  // compiler-optimization for speed

// Non-recursive CIC Filter (M=2, R=4) implementation, so two-stages of (followed by
```

```
down-sampling with factor 2):
// 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
// Non-recursive CIC Filter (M=4) implementation (for second-stage only):
// H1(z) = (1 + z^-1)^4 = 1 + 4*z^-1 + 6*z^-2 + 4*z^-3 + z^-4 = 1 + 6*z^-2 + z^-4 + (4
+ 4*z^-2) * z^-1 = FA(z) + FB(z) * z^-1;
// with down-sampling before stage translates into poly-phase components: FA(z) = 1 +
6*z^-1 + z^-2, FB(z) = 4 + 4*z^-1
// M=3 FA(z) = 1 + 3*z^-1, FB(z) = 3 + z^-1
// source: Lyons Understanding Digital Signal Processing 3rd edition 13.24.1

/* Basicdsp simulation:
# M=2 FA(z) = 1 + z^-1, FB(z) = 2
# M=3 FA(z) = 1 + 3*z^-1, FB(z) = 3 + z^-1
# M=4 FA(z) = 1 + 6*z^-1 + z^-2, FB(z) = 4 + 4*z^-1
samplerate=28000
x=x+1
clk1=mod1(x/2)*2
y=y+clk1
clk2=mod1(y/2)*2
#s1=clk1*fir(in, 1, 2, 1, 0)/16
#s2=clk2*fir(s1, 1, 0, 2, 0, 1, 0, 0)/16
#s1=clk1*fir(in, 1, 3, 3, 1, 0)/16
#s2=clk2*fir(s1, 1, 0, 3, 0, 3, 0, 1, 0, 0)/16
s1=clk1*fir(in, 1, 4, 6, 4, 1, 0)/16
s2=clk2*fir(s1, 1, 0, 4, 0, 6, 0, 4, 0, 1, 0, 0)/16
out=s2
 */

#define NEW_RX  1   // Faster (3rd-order) CIC stage, with simultanuous processing
capability
#ifdef NEW_RX
#define AF_OUT  1   // Enables audio output stage (can be disabled in conjunction with
CAT_STREAMING to safe memory)

static uint8_t tc = 0;
void process(int16_t i_ac2, int16_t q_ac2)
{
  static int16_t ac3;
#ifdef CAT_STREAMING
  //UCSR0B &= ~(TXCIE0);  // disable USART TX interrupts
  //while (!( UCSR0A & (1<<UDRE0)));  // wait for empty buffer
  if(cat_streaming){ uint8_t out = ac3 + 128; if(out == ';') out++; Serial.write(out);
}  //UDR0 = (uint8_t)(ac3 + 128);   // from:  https://www.xanthium.in/how-to-avr-
atmega328p-microcontroller-usart-uart-embedded-programming-avrgcc
#endif // CAT_STREAMING
#ifdef AF_OUT
  static int16_t ozd1, ozd2;  // Output stage
  if(_init){ ac3 = 0; ozd1 = 0; ozd2 = 0; _init = 0; } // hack: on first sample init
accumlators of further stages (to prevent instability)
  int16_t od1 = ac3 - ozd1; // Comb section
  ocomb = od1 - ozd2;
#endif //AF_OUT
#define OUTLET  1
#ifdef OUTLET
  if(tc++ == 0)   // prevent recursion
  //if(tc++ > 16)   // prevent recursion
#endif
  interrupts();  // hack, since slow_dsp process exceeds rx sample-time, allow
subsequent 7 interrupts for further rx sampling while processing, prevent nested
interrupts with tc
#ifdef AF_OUT
  ozd2 = od1;
  ozd1 = ac3;
#endif  //AF_OUT
  int16_t qh;
  {
    q_ac2 >>= att2;  // digital gain control
```

```
    static int16_t v[14];  // Process Q (down-sampled) samples
    // Hilbert transform, BasicDSP model:  outi= fir(inl,  0, 0, 0, 0, 0,  0,  0, 1,
0, 0,   0, 0,  0, 0, 0, 0); outq = fir(inr, 2, 0, 8, 0, 21, 0, 79, 0, -79, 0, -21, 0,
-8, 0, -2, 0) / 128;
    qh = ((v[0] - q_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[6] - v[8]) / 2; // Hilbert
transform, 43dB side-band rejection in 650..3400Hz (@8kSPS) when used in image-
rejection scenario; (Hilbert transform require 4 additional bits)
    //qh = ((v[0] - q_ac2) * 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 rejection
in 400..1900Hz (@4kSPS) when used in image-rejection scenario; (Hilbert transform
require 5 additional bits)
    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]
= v[7]; v[7] = v[8]; v[8] = v[9]; v[9] = v[10]; v[10] = v[11]; v[11] = v[12]; v[12] =
v[13]; v[13] = q_ac2;
  }
  i_ac2 >>= att2;  // digital gain control
  static int16_t v[7];  // Post processing I and Q (down-sampled) results
  i = i_ac2; q = q_ac2;   // tbd: this can be more efficient
  int16_t 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] = i_ac2;  // Delay to match Hilbert transform on Q branch
  ac3 = ac3 = slow_dsp(-i - qh);  //inverting I and Q helps dampening a feedback-loop
between PWM out and ADC inputs
#ifdef OUTLET
  tc--;
#endif
}

/*
// # M=3 .. = i0 + 3*(i2 + i3) + i1
int16_t i0, i1, i2, i3, i4, i5, i6, i7, i8;
int16_t q0, q1, q2, q3, q4, q5, q6, q7, q8;
#define M_SR  1

//#define EXPANDED_CIC
#ifdef EXPANDED_CIC
void sdr_rx_00(){         i0 = sdr_rx_common_i(); func_ptr = sdr_rx_01;   i4 = (i0 +
(i2 + i3) * 3 + i1) >> M_SR; }
void sdr_rx_02(){         i1 = sdr_rx_common_i(); func_ptr = sdr_rx_03;   i8 = (i4 +
(i6 + i7) * 3 + i5) >> M_SR; }
void sdr_rx_04(){         i2 = sdr_rx_common_i(); func_ptr = sdr_rx_05;   i5 = (i2 +
(i0 + i1) * 3 + i3) >> M_SR; }
void sdr_rx_06(){         i3 = sdr_rx_common_i(); func_ptr = sdr_rx_07; }
void sdr_rx_08(){         i0 = sdr_rx_common_i(); func_ptr = sdr_rx_09;   i6 = (i0 +
(i2 + i3) * 3 + i1) >> M_SR; }
void sdr_rx_10(){         i1 = sdr_rx_common_i(); func_ptr = sdr_rx_11;   i8 = (i6 +
(i4 + i5) * 3 + i7) >> M_SR; }
void sdr_rx_12(){         i2 = sdr_rx_common_i(); func_ptr = sdr_rx_13;   i7 = (i2 +
(i0 + i1) * 3 + i3) >> M_SR; }
void sdr_rx_14(){         i3 = sdr_rx_common_i(); func_ptr = sdr_rx_15; }
void sdr_rx_15(){         q0 = sdr_rx_common_q(); func_ptr = sdr_rx_00;   q4 = (q0 +
(q2 + q3) * 3 + q1) >> M_SR; }
void sdr_rx_01(){         q1 = sdr_rx_common_q(); func_ptr = sdr_rx_02;   q8 = (q4 +
(q6 + q7) * 3 + q5) >> M_SR; }
void sdr_rx_03(){         q2 = sdr_rx_common_q(); func_ptr = sdr_rx_04;   q5 = (q2 +
(q0 + q1) * 3 + q3) >> M_SR; }
void sdr_rx_05(){         q3 = sdr_rx_common_q(); func_ptr = sdr_rx_06; process(i8,
q8); }
void sdr_rx_07(){         q0 = sdr_rx_common_q(); func_ptr = sdr_rx_08;   q6 = (q0 +
(q2 + q3) * 3 + q1) >> M_SR; }
void sdr_rx_09(){         q1 = sdr_rx_common_q(); func_ptr = sdr_rx_10;   q8 = (q6 +
(q4 + q5) * 3 + q7) >> M_SR; }
void sdr_rx_11(){         q2 = sdr_rx_common_q(); func_ptr = sdr_rx_12;   q7 = (q2 +
(q0 + q1) * 3 + q3) >> M_SR; }
void sdr_rx_13(){         q3 = sdr_rx_common_q(); func_ptr = sdr_rx_14; process(i8,
q8); }
#else
void sdr_rx_00(){         i0 = sdr_rx_common_i(); func_ptr = sdr_rx_01;   i4 = (i0 +
(i2 + i3) * 3 + i1) >> M_SR; }
void sdr_rx_02(){         i1 = sdr_rx_common_i(); func_ptr = sdr_rx_03;   i8 = (i4 +
```

```
(i6 + i7) * 3 + i5) >> M_SR; }
void sdr_rx_04(){         i2 = sdr_rx_common_i(); func_ptr = sdr_rx_05;   i5 = (i2 +
(i0 + i1) * 3 + i3) >> M_SR; }
void sdr_rx_06(){         i3 = sdr_rx_common_i(); func_ptr = sdr_rx_07;   i6 = i4; i7 =
i5; q6 = q4; q7 = q5; }
void sdr_rx_07(){         q0 = sdr_rx_common_q(); func_ptr = sdr_rx_00;   q4 = (q0 +
(q2 + q3) * 3 + q1) >> M_SR; }
void sdr_rx_01(){         q1 = sdr_rx_common_q(); func_ptr = sdr_rx_02;   q8 = (q4 +
(q6 + q7) * 3 + q5) >> M_SR; }
void sdr_rx_03(){         q2 = sdr_rx_common_q(); func_ptr = sdr_rx_04;   q5 = (q2 +
(q0 + q1) * 3 + q3) >> M_SR; }
void sdr_rx_05(){         q3 = sdr_rx_common_q(); func_ptr = sdr_rx_06; process(i8,
q8); }
#endif
*/

// /*
static int16_t i_s0za1, i_s0za2, i_s0zb0, i_s0zb1, i_s1za1, i_s1za2, i_s1zb0, i_s1zb1;
static int16_t q_s0za1, q_s0za2, q_s0zb0, q_s0zb1, q_s1za1, q_s1za2, q_s1zb0, q_s1zb1,
q_ac2;

#define M_SR  1  // CIC N=3
void sdr_rx_00(){ int16_t ac = sdr_rx_common_i(); func_ptr = sdr_rx_01;  int16_t
i_s1za0 = (ac + (i_s0za1 + i_s0zb0) * 3 + i_s0zb1) >> M_SR; i_s0za1 = ac; int16_t ac2 =
(i_s1za0 + (i_s1za1 + i_s1zb0) * 3 + i_s1zb1); i_s1za1 = i_s1za0; process(ac2, q_ac2);
}
void sdr_rx_02(){ int16_t ac = sdr_rx_common_i(); func_ptr = sdr_rx_03;  i_s0zb1 =
i_s0zb0; i_s0zb0 = ac; }
void sdr_rx_04(){ int16_t ac = sdr_rx_common_i(); func_ptr = sdr_rx_05;  i_s1zb1 =
i_s1zb0; i_s1zb0 = (ac + (i_s0za1 + i_s0zb0) * 3 + i_s0zb1) >> M_SR; i_s0za1 = ac; }
void sdr_rx_06(){ int16_t ac = sdr_rx_common_i(); func_ptr = sdr_rx_07;  i_s0zb1 =
i_s0zb0; i_s0zb0 = ac; }
void sdr_rx_01(){ int16_t ac = sdr_rx_common_q(); func_ptr = sdr_rx_02;  q_s0zb1 =
q_s0zb0; q_s0zb0 = ac; }
void sdr_rx_03(){ int16_t ac = sdr_rx_common_q(); func_ptr = sdr_rx_04;  q_s1zb1 =
q_s1zb0; q_s1zb0 = (ac + (q_s0za1 + q_s0zb0) * 3 + q_s0zb1) >> M_SR; q_s0za1 = ac; }
void sdr_rx_05(){ int16_t ac = sdr_rx_common_q(); func_ptr = sdr_rx_06;  q_s0zb1 =
q_s0zb0; q_s0zb0 = ac; }
void sdr_rx_07(){ int16_t ac = sdr_rx_common_q(); func_ptr = sdr_rx_00;  int16_t
q_s1za0 = (ac + (q_s0za1 + q_s0zb0) * 3 + q_s0zb1) >> M_SR; q_s0za1 = ac; q_ac2 =
(q_s1za0 + (q_s1za1 + q_s1zb0) * 3 + q_s1zb1); q_s1za1 = q_s1za0; }
// */

/*
#define M_SR  0  // CIC N=2
void sdr_rx_00(){ int16_t ac = sdr_rx_common_i(); func_ptr = sdr_rx_01;  int16_t
i_s1za0 = (ac + i_s0za1 + i_s0zb0 * 2 + i_s0zb1) >> M_SR; i_s0za1 = ac; int16_t ac2 =
(i_s1za0 + i_s1za1 + i_s1zb0 * 2); i_s1za1 = i_s1za0; process(ac2, q_ac2); }
void sdr_rx_02(){ int16_t ac = sdr_rx_common_i(); func_ptr = sdr_rx_03;  i_s0zb0 = ac;
}
void sdr_rx_04(){ int16_t ac = sdr_rx_common_i(); func_ptr = sdr_rx_05;  i_s1zb0 = (ac
+ i_s0za1 + i_s0zb0 * 2) >> M_SR; i_s0za1 = ac; }
void sdr_rx_06(){ int16_t ac = sdr_rx_common_i(); func_ptr = sdr_rx_07;  i_s0zb0 = ac;
}
void sdr_rx_01(){ int16_t ac = sdr_rx_common_q(); func_ptr = sdr_rx_02;  q_s0zb0 = ac;
}
void sdr_rx_03(){ int16_t ac = sdr_rx_common_q(); func_ptr = sdr_rx_04;  q_s1zb0 = (ac
+ q_s0za1 + q_s0zb0 * 2) >> M_SR; q_s0za1 = ac; }
void sdr_rx_05(){ int16_t ac = sdr_rx_common_q(); func_ptr = sdr_rx_06;  q_s0zb0 = ac;
}
void sdr_rx_07(){ int16_t ac = sdr_rx_common_q(); func_ptr = sdr_rx_00;  int16_t
q_s1za0 = (ac + q_s0za1 + q_s0zb0 * 2 + q_s0zb1) >> M_SR; q_s0za1 = ac; q_ac2 =
(q_s1za0 + q_s1za1 + q_s1zb0 * 2); q_s1za1 = q_s1za0; }
*/

static int16_t ozi1, ozi2;

inline int16_t sdr_rx_common_q(){
  ADMUX = admux[0]; ADCSRA |= (1 << ADSC); int16_t ac = ADC - 511;
/*ozi2 = ozi1 + ozi2;          // Integrator section - needed?
```

```
  ozi1 = ocomb + ozi1;
  OCR1AL = min(max(128 - (ozi2>>5) + 128, 0), 255); */
  return ac;
}

inline int16_t sdr_rx_common_i()
{
  ADMUX = admux[1]; ADCSRA |= (1 << ADSC); int16_t adc = ADC - 511;
  static int16_t prev_adc;
  int16_t ac = (prev_adc + adc) / 2; prev_adc = adc;
#ifdef AF_OUT
  if(_init){ ocomb=0; ozi1 = 0; ozi2 = 0; } // hack
  ozi2 = ozi1 + ozi2;          // Integrator section
  ozi1 = ocomb + ozi1;
  OCR1AL = min(max((ozi2>>5) + 128, 0), 255);
#endif // AF_OUT
  return ac;
}


#else // OLD_RX    //Orginal 2nd-order CIC:
//#define M4  1  // Enable to enable M=4 on second-stage (better alias rejection)

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;

  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;  // Delay to match Hilbert transform on Q branch

        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 accumlators of further stages (to prevent instability)
#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;

  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];
  // Hilbert transform, BasicDSP model:  outi= fir(inl,  0, 0, 0, 0, 0,  0,  0, 1,   0,
0,   0, 0,  0, 0, 0, 0); outq = fir(inr, 2, 0, 8, 0, 21, 0, 79, 0, -79, 0, -21, 0, -8,
0, -2, 0) / 128;
        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[6] - v[8]) / 2; //
Hilbert transform, 43dB side-band rejection in 650..3400Hz (@8kSPS) when used in image-
rejection scenario; (Hilbert transform require 4 additional bits)
        //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 transform, 40dB side-band rejection
in 400..1900Hz (@4kSPS) when used in image-rejection scenario; (Hilbert transform
require 5 additional bits)
        for(uint8_t j = 0; j != 13; j++) v[j] = v[j + 1]; v[13] = ac2;
        //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] = v[7]; v[7] = v[8]; v[8] = v[9]; v[9] = v[10]; v[10] = v[11]; v[11] = v[12];
v[12] = v[13]; v[13] = ac2;
      }
      rx_state = 0; return;
```

```
    } else _z1 = _ac;
  } else z1 = ac;

  rx_state++;
}

inline void sdr_rx_common()
{
  static int16_t ozi1, ozi2;
  if(_init){ ocomb=0; ozi1 = 0; ozi2 = 0; } // hack
  // Output stage [25% CPU@R=4;Fs=62.5k]
#ifdef SECOND_ORDER_DUC
  ozi2 = ozi1 + ozi2;          // Integrator section
#endif
  ozi1 = ocomb + ozi1;
#ifdef SECOND_ORDER_DUC
  OCR1AL = min(max((ozi2>>5) + 128, 0), 255);  // OCR1AL = min(max((ozi2>>5) + ICR1L/2,
0), ICR1L);  // center and clip wrt PWM working range
#else
  OCR1AL = (ozi1>>5) + 128;
  OCR1AL = min(max((ozi1>>5) + 128, 0), 255);  // OCR1AL = min(max((ozi2>>5) + ICR1L/2,
0), ICR1L);  // center and clip wrt PWM working range
#endif
}
#endif //OLD_RX

#endif  //!SIMPLE_RX

#ifdef SIMPLE_RX
volatile uint8_t admux[3];
static uint8_t rx_state = 0;

static struct rx {
  int16_t z1;
  int16_t za1;
  int16_t _z1;
  int16_t _za1;
} rx_inst[2];

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
    OCR1AL = min(max((ozi2>>5) + 128, 0), 255);  // OCR1AL = min(max((ozi2>>5) +
ICR1L/2, 0), ICR1L);  // center and clip wrt PWM working range
    #else
    OCR1AL = (ozi1>>5) + 128;
    //OCR1AL = min(max((ozi1>>5) + 128, 0), 255);  // OCR1AL = min(max((ozi2>>5) +
ICR1L/2, 0), ICR1L);  // center and clip wrt PWM working range
```

```
    #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 transform on Q branch

        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 accumlators 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) * 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 rejection
in 400..1900Hz (@4kSPS) when used in image-rejection scenario; (Hilbert transform
require 5 additional bits)
        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++;
}

#endif //SIMPLE_RX

ISR(TIMER2_COMPA_vect)  // Timer2 COMPA interrupt
{
  func_ptr();

}
```

```
#pragma GCC pop_options  // end of DSP section

void adc_start(uint8_t adcpin, bool ref1v1, uint32_t fs)
{
  DIDR0 |= (1 << adcpin); // disable digital input
  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);  // If reflvl == true, set
AREF=1.1V (Internal ref); otherwise AREF=AVCC=(5V)
  ADCSRA |= ((uint8_t)log2((uint8_t)(F_CPU / 13 / fs))) & 0x07;  // ADC Prescaler (for
normal conversions non-auto-triggered): ADPS = log2(F_CPU / 13 / Fs) - 1; ADSP=0..7
resulting in resp. conversion rate of 1536, 768, 384, 192, 96, 48, 24, 12 kHz
  //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 sleep is better alternative (keeping clkIO as an active clock domain)
  set_sleep_mode(SLEEP_MODE_IDLE);
  sleep_enable();
#endif
}

void adc_stop()
{
  //ADCSRA &= ~(1 << ADATE); // disable auto trigger
  ADCSRA &= ~(1 << ADIE);  // disable interrupts when measurement complete
  ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);    // 128 prescaler for 9.6kHz
#ifdef ADC_NR
  sleep_disable();
#endif
  ADMUX = (1 << REFS0);  // restore reference voltage AREF (5V)
}

void timer1_start(uint32_t fs)
{  // Timer 1: OC1A and OC1B in PWM mode
  TCCR1A = 0;
  TCCR1B = 0;
  TCCR1A |= (1 << COM1A1) | (1 << COM1B1) | (1 << WGM11); // Clear OC1A/OC1B on compare
match, set OC1A/OC1B at BOTTOM (non-inverting mode)
  TCCR1B |= (1 << CS10) | (1 << WGM13) | (1 << WGM12); // Mode 14 - Fast PWM;  CS10:
clkI/O/1 (No prescaling)
  ICR1H = 0x00;
  ICR1L = min(255, (float)F_CPU / (float)fs - 0.5);  // PWM value range (fs>78431):
Fpwm = F_CPU / [Prescaler * (1 + TOP)]
  //TCCR1A |= (1 << COM1A1) | (1 << COM1B1) | (1 << WGM10); // Clear OC1A/OC1B on
compare match, set OC1A/OC1B at BOTTOM (non-inverting mode)
  //TCCR1B |= (1 << CS10) | (1 << WGM12); // Mode 5 - Fast PWM, 8-bit;  CS10: clkI/O/1
(No prescaling)
  OCR1AH = 0x00;
  OCR1AL = 0x00;  // OC1A (SIDETONE) PWM duty-cycle (span defined by ICR).
  OCR1BH = 0x00;
  OCR1BL = 0x00;  // OC1B (KEY_OUT) PWM duty-cycle (span defined by ICR).
}

void timer1_stop()
{
  OCR1AL = 0x00;
  OCR1BL = 0x00;
}

void timer2_start(uint32_t fs)
{  // Timer 2: interrupt mode
  ASSR &= ~(1 << AS2);  // Timer 2 clocked from CLK I/O (like Timer 0 and 1)
  TCCR2A = 0;
  TCCR2B = 0;
  TCNT2 = 0;
  TCCR2A |= (1 << WGM21); // WGM21: Mode 2 - CTC (Clear Timer on Compare Match)
```

```
  TCCR2B |= (1 << CS22);  // Set C22 bits for 64 prescaler
  TIMSK2 |= (1 << OCIE2A);  // enable timer compare interrupt TIMER2_COMPA_vect
  uint8_t ocr = (((float)F_CPU / (float)64) / (float)fs + 0.5) - 1;   // OCRn = (F_CPU
/ pre-scaler / fs) - 1;
  OCR2A = ocr;
}

void timer2_stop()
{ // Stop Timer 2 interrupt
  TIMSK2 &= ~(1 << OCIE2A);  // disable timer compare interrupt
  delay(1);  // wait until potential in-flight interrupts are finished
}

////////////////////////////////////////////////////////////////////////////////////
//////////////////////
// Below a radio-specific implementation based on the above components (seperation of
concerns)
//
// Feel free to replace it with your own custom radio implementation :-)

void inline lcd_blanks(){ lcd.print(F("         ")); }

#define N_FONTS  8
const byte fonts[N_FONTS][8] PROGMEM = {
{ 0b01000,  // 1; logo
  0b00100,
  0b01010,
  0b00101,
  0b01010,
  0b00100,
  0b01000,
  0b00000 },
{ 0b00000,  // 2; s-meter, 0 bars
  0b00000,
  0b00000,
  0b00000,
  0b00000,
  0b00000,
  0b00000,
  0b00000 },
{ 0b10000,  // 3; s-meter, 1 bars
  0b10000,
  0b10000,
  0b10000,
  0b10000,
  0b10000,
  0b10000,
  0b10000 },
{ 0b10000,  // 4; s-meter, 2 bars
  0b10000,
  0b10100,
  0b10100,
  0b10100,
  0b10100,
  0b10100,
  0b10100 },
{ 0b10000,  // 5; s-meter, 3 bars
  0b10000,
  0b10101,
  0b10101,
  0b10101,
  0b10101,
  0b10101,
  0b10101 },
{ 0b01100,  // 6; vfo-a
  0b10010,
  0b11110,
  0b10010,
  0b10010,
  0b00000,
```

```
  0b00000,
  0b00000 },
{ 0b11100,  // 7; vfo-b
  0b10010,
  0b11100,
  0b10010,
  0b11100,
  0b00000,
  0b00000,
  0b00000 },
{ 0b00000,  // 8; TBD
  0b00000,
  0b00000,
  0b00000,
  0b00000,
  0b00000,
  0b00000,
  0b00000 }
};

int analogSafeRead(uint8_t pin, bool ref1v1 = false)
{  // performs classical analogRead with default Arduino sample-rate and analog
reference setting; restores previous settings
  noInterrupts();
  for(;!(ADCSRA & (1 << ADIF)););  // wait until (a potential previous) ADC conversion
is completed
  uint8_t adcsra = ADCSRA;
  uint8_t admux = ADMUX;
  ADCSRA &= ~(1 << ADIE);  // disable interrupts when measurement complete
  ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);    // 128 prescaler for 9.6kHz
  if(ref1v1) ADMUX &= ~(1 << REFS0);  // restore reference voltage AREF (1V1)
  else ADMUX = (1 << REFS0);  // restore reference voltage AREF (5V)
  delay(1);  // settle
  int val = analogRead(pin);
  ADCSRA = adcsra;
  ADMUX = admux;
  interrupts();
  return val;
}


uint16_t analogSampleMic()
{
  uint16_t adc;
  noInterrupts();
  ADCSRA = (1 << ADEN) | ((uint8_t)log2((uint8_t)(F_CPU / 13 / (192307/1)))) & 0x07;
// hack: faster conversion rate necessary for VOX

  if((dsp_cap == SDR) && (vox_thresh >= 32)) digitalWrite(RX, LOW);  // disable RF
input, only for SDR mod and with low VOX threshold
  //si5351.SendRegister(SI_CLK_OE, 0b11111111); // CLK2_EN=0, CLK1_EN,CLK0_EN=0
  uint8_t oldmux = ADMUX;
  for(;!(ADCSRA & (1 << ADIF)););  // wait until (a potential previous) ADC conversion
is completed
  ADMUX = admux[2];  // set MUX for next conversion
  ADCSRA |= (1 << ADSC);    // start next ADC conversion
  for(;!(ADCSRA & (1 << ADIF)););  // wait until ADC conversion is completed
  ADMUX = oldmux;
  if((dsp_cap == SDR) && (vox_thresh >= 32)) digitalWrite(RX, HIGH);  // enable RF
input, only for SDR mod and with low VOX threshold
  //si5351.SendRegister(SI_CLK_OE, 0b11111100); // CLK2_EN=0, CLK1_EN,CLK0_EN=1
  adc = ADC;
  interrupts();
  return adc;
}

volatile bool change = true;
volatile int32_t freq = 7035000;
static int32_t vfo[] = { 7035000, 14055000 };
static uint8_t vfomode[] = { USB, USB };
```

```
enum vfo_t { VFOA=0, VFOB=1, SPLIT=2 };
volatile uint8_t vfosel = VFOA;
volatile int16_t rit = 0;

// We measure the average amplitude of the signal (see slow_dsp()) but the S-meter
should be based on RMS value.
// So we multiply by 0.707/0.639 in an attempt to roughly compensate, although that
only really works if the input
// is a sine wave
uint8_t smode = 1;
uint32_t max_absavg256 = 0;
float dbm;

static int16_t smeter_cnt = 0;

float smeter(float ref = 0)
{
  max_absavg256 = max(_absavg256, max_absavg256); // peak

  if((smode) && ((++smeter_cnt % 2048) == 0)){   // slowed down display slightly
    float rms;
    if(dsp_cap == SDR) rms = (float)max_absavg256 * 1.1 * (float)(1 << att2) / (256.0 *
1024.0 * (float)R * 8.0 * 500.0 * 1.414 / 0.707);   // 1 rx gain stage: rmsV = ADC
value * AREF / [ADC DR * processing gain * receiver gain * "RMS compensation"]
    else               rms = (float)max_absavg256 * 5.0 * (float)(1 << att2) / (256.0 *
1024.0 * (float)R * 2.0 * 100.0 * 120.0 / 1.750);
    dbm = (10.0 * log10((rms * rms) / 50.0) + 30.0) - ref; //from rmsV to dBm at 50R

    lcd.noCursor();
    if(smode == 1){ // dBm meter
      lcd.setCursor(9, 0); lcd.print((int16_t)dbm); lcd.print(F("dBm "));
    }
    if(smode == 2){ // S-meter
      uint8_t s = (dbm < -63) ? ((dbm - -127) / 6) : (((uint8_t)(dbm - -73)) / 10) *
10;  // dBm to S (modified to work correctly above S9)
      lcd.setCursor(14, 0); if(s < 10){ lcd.print('S'); } lcd.print(s);
    }
    if(smode == 3){ // S-bar
      int8_t s = (dbm < -63) ? ((dbm - -127) / 6) : (((uint8_t)(dbm - -73)) / 10) * 10;
// dBm to S (modified to work correctly above S9)
      char tmp[5];
      for(uint8_t i = 0; i != 4; i++){ tmp[i] = max(2, min(5, s + 1)); s = s - 3; }
tmp[4] = 0;
      lcd.setCursor(12, 0); lcd.print(tmp);
    }
#ifdef CW_DECODER
    if(smode == 4){ // wpm-indicator
      lcd.setCursor(14, 0); if(mode == CW) lcd.print(wpm); lcd.print("  ");
    }
#endif  //CW_DECODER


    stepsize_showcursor();
    max_absavg256 /= 2;  // Implement peak hold/decay for all meter types
  }
  return dbm;
}

void start_rx()
{
  _init = 1;
  rx_state = 0;
  func_ptr = sdr_rx_00;  //enable RX DSP/SDR
  adc_start(2, true, F_ADC_CONV*4); admux[2] = ADMUX;  // Note that conversion-rate for
TX is factors more
  if(dsp_cap == SDR){
//#define SWAP_RX_IQ 1    // Swap I/Q ADC inputs, flips RX sideband
#ifdef SWAP_RX_IQ
    adc_start(1, !(att == 1)/*true*/, F_ADC_CONV); admux[0] = ADMUX;
    adc_start(0, !(att == 1)/*true*/, F_ADC_CONV); admux[1] = ADMUX;
```

```
#else
    adc_start(0, !(att == 1)/*true*/, F_ADC_CONV); admux[0] = ADMUX;
    adc_start(1, !(att == 1)/*true*/, F_ADC_CONV); admux[1] = ADMUX;
#endif //SWAP_RX_IQ
  } 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
}

int16_t _centiGain = 0;

uint8_t txdelay = 0;
uint8_t semi_qsk = false;
uint32_t semi_qsk_timeout = 0;

void switch_rxtx(uint8_t tx_enable){
  TIMSK2 &= ~(1 << OCIE2A);  // disable timer compare interrupt
  //delay(1);
  delayMicroseconds(20); // wait until potential RX interrupt is finalized
  noInterrupts();
#ifdef TX_DELAY
#ifdef SEMI_QSK
  if(!(semi_qsk_timeout))
#endif
    if((txdelay) && (tx_enable) && (!(tx)) && (!(practice))){  // key-up TX relay in
advance before actual transmission
      digitalWrite(RX, LOW); // TX (disable RX)
#ifdef NTX
      digitalWrite(NTX, LOW);  // TX (enable TX)
#endif //NTX
#ifdef PTX
      digitalWrite(PTX, HIGH);  // TX (enable TX)
#endif //PTX
      lcd.setCursor(15, 1); lcd.print('D');  // note that this enables interrupts
again.
      interrupts();    //hack.. to allow delay()
      delay(F_MCU * txdelay /16000000);
      noInterrupts();  //end of hack
    }
#endif //TX_DELAY
  tx = tx_enable;
  if(tx_enable){  // tx
    _centiGain = centiGain;  // backup AGC setting
#ifdef SEMI_QSK
    semi_qsk_timeout = 0;
#endif
    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 {  // rx
    if((mode == CW) && (!(semi_qsk_timeout))){
#ifdef SEMI_QSK
#ifdef KEYER
      semi_qsk_timeout = millis() + ditTime * 8;
#else
      semi_qsk_timeout = millis() + 8 * 8;  // no keyer? assume dit-time of 20 WPM
#endif //KEYER
#endif //SEMI_QSK
      if(semi_qsk) func_ptr = dummy; else func_ptr = sdr_rx_00;
    } else {
      centiGain = _centiGain;  // restore AGC setting
#ifdef SEMI_QSK
      semi_qsk_timeout = 0;
```

```
#endif
      func_ptr = sdr_rx_00;
    }
  }
  if((!dsp_cap) && (!tx_enable) && vox) func_ptr = dummy; //hack: for SSB mode, disable
dsp_rx during vox mode enabled as it slows down the vox loop too much!
  interrupts();
  if(tx_enable) ADMUX = admux[2];
  else _init = 1;
  rx_state = 0;
#ifdef CW_DECODER
  if((cwdec) && (mode == CW)){ filteredstate = tx_enable; dec2(); }
#endif  //CW_DECODER

  if(tx_enable){ // tx
    if(practice){
      digitalWrite(RX, LOW); // TX (disable RX)
      lcd.setCursor(15, 1); lcd.print('P');
      si5351.SendRegister(SI_CLK_OE, 0b11111111); // CLK2_EN,CLK1_EN,CLK0_EN=0
      // Do not enable PWM (KEY_OUT), do not enble CLK2
    } else
    {
      digitalWrite(RX, LOW); // TX (disable RX)
#ifdef NTX
      digitalWrite(NTX, LOW);  // TX (enable TX)
#endif //NTX
#ifdef PTX
      digitalWrite(PTX, HIGH);  // TX (enable TX)
#endif //PTX
      lcd.setCursor(15, 1); lcd.print('T');
      if(mode == CW){ si5351.freq_calc_fast(-cw_offset); si5351.SendPLLRegisterBulk();
} // for CW, TX at freq
#ifdef RIT_ENABLE
      else if(rit){ si5351.freq_calc_fast(0); si5351.SendPLLRegisterBulk(); }
#endif //RIT_ENABLE
#ifdef TX_CLK0_CLK1
      si5351.SendRegister(SI_CLK_OE, 0b11111100); // CLK2_EN=0, CLK1_EN,CLK0_EN=1
#else
      si5351.SendRegister(SI_CLK_OE, 0b11111011); // CLK2_EN=1, CLK1_EN,CLK0_EN=0
#endif  //TX_CLK0_CLK1
      OCR1AL = 0x80; // make sure SIDETONE is set at 2.5V
      if((!mox) && (mode != CW)) TCCR1A &= ~(1 << COM1A1); // disable SIDETONE, prevent
interference during SSB TX
      TCCR1A |= (1 << COM1B1);  // enable KEY_OUT PWM
#ifdef _SERIAL
      if(cat_active){ DDRC &= ~(1<<2); } // disable PC2, so that ADC2 can be used as
mic input
#endif
    }
  } else {  // rx
#ifdef KEY_CLICK
      if(OCR1BL != 0) {
       for(uint16_t i = 0; i != 31; i++) {   // ramp down of amplitude: soft falling
edge to prevent key clicks
         OCR1BL = lut[pgm_read_byte_near(ramp[i])];
          delayMicroseconds(60);
       }
      }
#endif //KEY_CLICK
      TCCR1A |= (1 << COM1A1);  // enable SIDETONE (was disabled to prevent
interference during ssb tx)
      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%
#ifdef QUAD
#ifdef TX_CLK0_CLK1
      si5351.SendRegister(16, 0x0f);  // disable invert on CLK0
      si5351.SendRegister(17, 0x0f);  // disable invert on CLK1
#else
      si5351.SendRegister(18, 0x0f);  // disable invert on CLK2
```

```
#endif  //TX_CLK0_CLK1
#endif //QUAD
      si5351.SendRegister(SI_CLK_OE, 0b11111100); // CLK2_EN=0, CLK1_EN,CLK0_EN=1
#ifdef SEMI_QSK
      if((!semi_qsk_timeout) || (!semi_qsk))   // enable RX when no longer in semi-qsk
phase; so RX and NTX/PTX outputs are switching only when in RX mode
#endif //SEMI_QSK
      {
        digitalWrite(RX, !(att == 2)); // RX (enable RX when attenuator not on)
#ifdef NTX
        digitalWrite(NTX, HIGH);  // RX (disable TX)
#endif //NTX
#ifdef PTX
        digitalWrite(PTX, LOW);   // TX (disable TX)
#endif //PTX
      }
#ifdef RIT_ENABLE
      si5351.freq_calc_fast(rit); si5351.SendPLLRegisterBulk();  // restore original
PLL RX frequency
#endif //RIT_ENABLE

      lcd.setCursor(15, 1); lcd.print((vox) ? 'V' : 'R');
#ifdef _SERIAL
      if(!vox) if(cat_active){ DDRC |= (1<<2); } // enable PC2, so that ADC2 is pulled-
down so that CAT TX is not disrupted via mic input
#endif
  }
  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
}

uint8_t rx_ph_q = 90;
uint8_t prev_bandval = 3;
uint8_t bandval = 3;
#define N_BANDS 11

#ifdef CW_FREQS_QRP
uint32_t band[N_BANDS] = { /*472000,*/ 1810000, 3560000, 5351500, 7030000, 10106000,
14060000, 18096000, 21060000, 24906000, 28060000, 50096000/*, 70160000, 144060000*/ };
// CW QRP freqs
#else
#ifdef CW_FREQS_FISTS
uint32_t band[N_BANDS] = { /*472000,*/ 1818000, 3558000, 5351500, 7028000, 10118000,
14058000, 18085000, 21058000, 24908000, 28058000, 50058000/*, 70158000, 144058000*/ };
// CW FISTS freqs
#else
uint32_t band[N_BANDS] = { /*472000,*/ 1840000, 3573000, 5357000, 7074000, 10136000,
14074000, 18100000, 21074000, 24915000, 28074000, 50313000/*, 70101000, 144125000*/ };
// FT8 freqs
#endif
#endif

enum step_t { STEP_10M, STEP_1M, STEP_500k, STEP_100k, STEP_10k, STEP_1k, STEP_500,
STEP_100, STEP_10, STEP_1 };
uint32_t stepsizes[10] = { 10000000, 1000000, 500000, 100000, 10000, 1000, 500, 100,
10, 1 };
volatile uint8_t stepsize = STEP_1k;
uint8_t prev_stepsize[] = { STEP_1k, STEP_500 }; //default stepsize for resp. SSB, CW

void process_encoder_tuning_step(int8_t steps)
{
  int32_t stepval = stepsizes[stepsize];
  //if(stepsize < STEP_100) freq %= 1000; // when tuned and stepsize > 100Hz then
forget fine-tuning details
  if(rit){
    rit += steps * stepval;
    rit = max(-9999, min(9999, rit));
  } else {
    freq += steps * stepval;
```

```
    freq = max(1, min(999999999, freq));
  }
  change = true;
}

void stepsize_showcursor()
{
  lcd.setCursor(stepsize+1, 1);  // display stepsize with cursor
  lcd.cursor();
}

void stepsize_change(int8_t val)
{
  stepsize += val;
  if(stepsize < STEP_1M) stepsize = STEP_10;
  if(stepsize > STEP_10) stepsize = STEP_1M;
  if(stepsize == STEP_500k) stepsize += val;
  stepsize_showcursor();
}

void powerDown()
{ // Reduces power from 110mA to 70mA (back-light on) or 30mA (back-light off),
remaining current is probably opamp quiescent current
  lcd.setCursor(0, 0); lcd.print(F("Power-off"));lcd_blanks();
  lcd.setCursor(0, 1); lcd.print(F("Adios 73 :-)")); lcd_blanks();
  MCUSR = ~(1<<WDRF);  // MSY be done before wdt_disable()
  wdt_disable();   // WDTON Fuse High bit need to be 1 (0xD1), if NOT it will override
and set WDE=1; WDIE=0, meaning MCU will reset when watchdog timer is zero, and this
seems to happen when wdt_disable() is called
  timer2_stop();
  timer1_stop();
  adc_stop();
  si5351.powerDown();
  delay(2000);
  // Disable external interrupts INT0, INT1, Pin Change
  PCICR = 0;
  PCMSK0 = 0;
  PCMSK1 = 0;
  PCMSK2 = 0;
  // Disable internal interrupts
  TIMSK0 = 0;
  TIMSK1 = 0;
  TIMSK2 = 0;
  WDTCSR = 0;
  // Enable BUTTON Pin Change interrupt
  *digitalPinToPCMSK(BUTTONS) |= (1<<digitalPinToPCMSKbit(BUTTONS));
  *digitalPinToPCICR(BUTTONS) |= (1<<digitalPinToPCICRbit(BUTTONS));
  // Power-down sub-systems
  PRR = 0xff;
  lcd.noDisplay();
  PORTD &= ~0x08; // disable backlight

  set_sleep_mode(SLEEP_MODE_PWR_DOWN);
  sleep_enable();
  interrupts();
  sleep_bod_disable();
  //MCUCR |= (1<<BODS) | (1<<BODSE);  // turn bod off by settings BODS, BODSE; note
BODS is reset after three clock-cycles, so quickly go to sleep before it is too late
  //MCUCR &= ~(1<<BODSE);  // must be done right before sleep
  sleep_cpu();  // go to sleep mode, wake-up by either INT0, INT1, Pin Change, TWI Addr
Match, WDT, BOD
  sleep_disable();

  //void(* reset)(void) = 0; reset();   // soft reset by calling reset vector (does not
reset registers to defaults)
  do { wdt_enable(WDTO_15MS); for(;;); } while(0);  // soft reset by trigger watchdog
timeout
}

void show_banner(){
```

```
  lcd.setCursor(0, 0);
  lcd_blanks(); lcd_blanks();
}

const char* vfosel_label[] = { "A", "B"/*, "Split"*/ };
const char* mode_label[5] = { "LSB", "USB", "CW ", "FM ", "AM " };

inline void display_vfo(int32_t f){
  lcd.setCursor(0, 1);
  lcd.print((rit) ? ' ' : ((vfosel%2)|((vfosel==SPLIT) & tx)) ? '\x07' : '\x06');  //
RIT, VFO A/B

  int32_t scale=10e6;
  if(rit){
    f = rit;
    scale=1e3;  // RIT frequency
    lcd.print(F("RIT ")); lcd.print(rit < 0 ? '-' : '+');
  } else {
    if(f/scale == 0){ lcd.print(' '); scale/=10; }  // Initial space instead of zero
  }
  for(; scale!=1; f%=scale, scale/=10){
    lcd.print(abs(f/scale));
    if(scale == (int32_t)1e3 || scale == (int32_t)1e6) lcd.print(',');  // Thousands
separator
  }

  lcd.print(' '); lcd.print(mode_label[mode]); lcd.print(' ');
  lcd.setCursor(15, 1); lcd.print((vox) ? 'V' : 'R');
}

volatile uint8_t event;
//volatile uint8_t menumode = 0;  // 0=not in menu, 1=selects menu item, 2=selects
parameter value
volatile uint8_t prev_menumode = 0;
volatile int8_t menu = 0;  // current parameter id selected in menu

#define pgm_cache_item(addr, sz) byte _item[sz]; memcpy_P(_item, addr, sz);  // copy
array item from PROGMEM to SRAM
#define get_version_id() ((VERSION[0]-'1') * 2048 + ((VERSION[2]-'0')*10 +
(VERSION[3]-'0')) * 32 +  ((VERSION[4]) ? (VERSION[4] - 'a' + 1) : 0) * 1)  // converts
VERSION string with (fixed) format "9.99z" into uint16_t (max. values shown here, z may
be removed)

uint8_t eeprom_version;
#define EEPROM_OFFSET 0x150  // avoid collision with QCX settings, overwrites text
settings though
int eeprom_addr;

// Support functions for parameter and menu handling
enum action_t { UPDATE, UPDATE_MENU, NEXT_MENU, LOAD, SAVE, SKIP, NEXT_CH };

// output menuid in x.y format
void printmenuid(uint8_t menuid){
  static const char seperator[] = {'.', ' '};
  uint8_t ids[] = {(uint8_t)(menuid >> 4), (uint8_t)(menuid & 0xF)};
  for(int i = 0; i < 2; i++){
    uint8_t id = ids[i];
    if(id >= 10){
      id -= 10;
      lcd.print('1');
    }
    lcd.print(char('0' + id));
    lcd.print(seperator[i]);
  }
}

void printlabel(uint8_t action, uint8_t menuid, const __FlashStringHelper* label){
  if(action == UPDATE_MENU){
    lcd.setCursor(0, 0);
    printmenuid(menuid);
```

```
    lcd.print(label); lcd_blanks(); lcd_blanks();
    lcd.setCursor(0, 1); // value on next line
    if(menumode >= 2) lcd.print('>');
  } else { // UPDATE (not in menu)
    lcd.setCursor(0, 1); lcd.print(label); lcd.print(F(": "));
  }
}

void actionCommon(uint8_t action, uint8_t *ptr, uint8_t size){
  uint8_t n;
  switch(action){
    case LOAD:
      //for(n = size; n; --n) *ptr++ = eeprom_read_byte((uint8_t *)eeprom_addr++);
      eeprom_read_block((void *)ptr, (const void *)eeprom_addr, size);
      break;
    case SAVE:
      //noInterrupts();
      //for(n = size; n; --n){ wdt_reset(); eeprom_write_byte((uint8_t *)eeprom_addr++,
*ptr++); }
      eeprom_write_block((const void *)ptr, (void *)eeprom_addr, size);
      //interrupts();
      break;
    case SKIP:
      //eeprom_addr += size;
      break;
  }
  eeprom_addr += size;
}

template<typename T> void paramAction(uint8_t action, volatile T& value, uint8_t
menuid, const __FlashStringHelper* label, const char* enumArray[], int32_t _min,
int32_t _max, bool continuous){
  switch(action){
    case UPDATE:
    case UPDATE_MENU:
      if(((int32_t)value + encoder_val) < _min) value = (continuous) ? _max : _min;
      else if(((int32_t)value + encoder_val) > _max) value = (continuous) ? _min :
_max;
      else value = (int32_t)value + encoder_val;
      encoder_val = 0;

      printlabel(action, menuid, label);  // print normal/menu label
      if(enumArray == NULL){  // print value
        if((_min < 0) && (value >= 0)) lcd.print('+');  // add + sign for positive
values, in case negative values are supported
        lcd.print(value);
      } else {
        lcd.print(enumArray[value]);
      }
      lcd_blanks(); lcd_blanks();
      //if(action == UPDATE) paramAction(SAVE, value, menuid, label, enumArray, _min,
_max, continuous, init_val);
      break;
    default:
      actionCommon(action, (uint8_t *)&value, sizeof(value));
      break;
  }
}

#ifdef MENU_STR
static uint8_t pos = 0;
void paramAction(uint8_t action, char* value, uint8_t menuid, const
__FlashStringHelper* label, uint8_t size){
  const uint8_t _min = ' '; const uint8_t _max = 'Z';
  switch(action){
    case NEXT_CH:
      if(pos < size) pos++;  // allow to go to next character when string size allows
and when current character is not string end
      action = UPDATE_MENU; //fall-through next case
    case UPDATE:
```

```
    case UPDATE_MENU:
      if(menumode != 3) pos = 0;
      if(menumode == 2) menumode = 3; // hack: for strings enter in edit mode
      if(((value[pos] + encoder_val) < _min) || ((value[pos] + encoder_val) == 0))
value[pos] = _min;
      else if((value[pos] + encoder_val) > _max) value[pos] = _max;
      else value[pos] = value[pos] + encoder_val;
      encoder_val = 0;

      printlabel(action, menuid, label);  // print normal/menu label
      for(int i = 0; i != 13; i++){ char ch = value[(pos / 8) * 8 + i]; if(ch)
lcd.print(ch); else break; } // print value
      //lcd.print(&value[(pos / 8) * 8]); // print value
      lcd.print('\x01');  // print terminator
      lcd_blanks();
      lcd.setCursor((pos % 8) + (menumode >= 2), 1); lcd.cursor();
      break;
    case SAVE:
      for(uint8_t i = size; i > 0; i--){
        if((value[i-1] == ' ') || (value[i-1] == 0)) value[i-1] = 0;  // remove
trailing spaces
        else break; // stop once content found
      }
      //fall-through next case
    default:
      actionCommon(action, (uint8_t *)value, size);
      break;
  }
}
#endif //MENU_STR

static uint32_t save_event_time = 0;
static uint8_t vox_tx = 0;
static uint8_t vox_sample = 0;
static uint16_t vox_adc = 0;

static uint8_t pwm_min = 0;    // PWM value for which PA reaches its minimum: 29 when
C31 installed;   0 when C31 removed;   0 for biasing BS170 directly

static uint8_t pwm_max = 128;  // PWM value for which PA reaches its maximum:
128 for biasing BS170 directly


const char* offon_label[2] = {"OFF", "ON"};
#if(F_MCU > 16000000)
const char* filt_label[N_FILT+1] = { "Full", "3000", "2400", "1800", "500", "200",
"100", "50" };
#else
const char* filt_label[N_FILT+1] = { "Full", "2400", "2000", "1500", "500", "200",
"100", "50" };
#endif
const char* band_label[N_BANDS] = { "160m", "80m", "60m", "40m", "30m", "20m", "17m",
"15m", "12m", "10m", "6m" };
const char* stepsize_label[] = { "10M", "1M", "0.5M", "100k", "10k", "1k", "0.5k",
"100", "10", "1" };
const char* att_label[] = { "0dB", "-13dB", "-20dB", "-33dB", "-40dB", "-53dB",
"-60dB", "-73dB" };
const char* smode_label[] = { "OFF", "dBm", "S", "S-bar", "wpm" };
const char* cw_tone_label[] = { "700", "600" };
#ifdef KEYER
const char* keyer_mode_label[] = { "Iambico A", "Iambico B","Vertical" };
#endif
#ifdef CW_LEARN
const char* learn_mode_label[] = { "OFF", "LEARN CW","PRACTICA CW" };
#endif
const char* agc_label[] = { "OFF", "Rapido", "Lento" };

#define _N(a) sizeof(a)/sizeof(a[0])

#define N_PARAMS 44  // number of (visible) parameters
```

```
#define N_ALL_PARAMS (N_PARAMS+5)  // number of parameters

enum params_t {_NULL, VOLUME, MODE, FILTER, BAND, STEP, VFOSEL, RIT, AGC, NR, ATT,
ATT2, SMETER, CWDEC, CWTONE, LEARN_MODE, CWOFF, SEMIQSK, KEY_WPM, KEY_MODE, KEY_PIN,
KEY_TX, VOX, VOXGAIN, TXDELAY, MOX, CWINTERVAL, CWMSG1, CWMSG2, CWMSG3, CWMSG4, CWMSG5,
CWMSG6, DRIVE, PWM_MIN, PWM_MAX, SIFXTAL, IQ_ADJ, CALIB, SR, CPULOAD, PARAM_A, PARAM_B,
PARAM_C, BACKL, FREQA, FREQB, MODEA, MODEB, VERS, ALL=0xff};

int8_t paramAction(uint8_t action, uint8_t id = ALL)  // list of parameters
{
  if((action == SAVE) || (action == LOAD)){
    eeprom_addr = EEPROM_OFFSET;
    for(uint8_t _id = 1; _id < id; _id++) paramAction(SKIP, _id);
  }
  if(id == ALL) for(id = 1; id != N_ALL_PARAMS+1; id++) paramAction(action, id);  //
for all parameters

  switch(id){    // Visible parameters
    case VOLUME:  paramAction(action, volume, 0x11, F("Volumen"), NULL, -1, 16, false);
break;
    case MODE:    paramAction(action, mode, 0x12, F("Modo"), mode_label, 0,
_N(mode_label) - 1, false); break;
    case FILTER:  paramAction(action, filt, 0x13, F("Filtro BW"), filt_label, 0,
_N(filt_label) - 1, false); break;
    case BAND:    paramAction(action, bandval, 0x14, F("Banda"), band_label, 0,
_N(band_label) - 1, false); break;
    case STEP:    paramAction(action, stepsize, 0x15, F("Tune Rate"), stepsize_label,
0, _N(stepsize_label) - 1, false); break;
    case VFOSEL:  paramAction(action, vfosel, 0x16, F("Modo VFO"), vfosel_label, 0,
_N(vfosel_label) - 1, false); break;
#ifdef RIT_ENABLE
    case RIT:     paramAction(action, rit, 0x17, F("RIT"), offon_label, 0, 1, false);
break;
#endif
#ifdef FAST_AGC
    case AGC:     paramAction(action, agc, 0x18, F("AGC"), agc_label, 0, _N(agc_label)
- 1, false); break;
#else
    case AGC:     paramAction(action, agc, 0x18, F("AGC"), offon_label, 0, 1, false);
break;
#endif // FAST_AGC
    case NR:      paramAction(action, nr, 0x19, F("NR"), NULL, 0, 8, false); break;
    case ATT:     paramAction(action, att, 0x1A, F("ATT"), att_label, 0, 7, false);
break;
    case ATT2:    paramAction(action, att2, 0x1B, F("ATT2"), NULL, 0, 16, false);
break;
    case SMETER:  paramAction(action, smode, 0x1C, F("S-meter"), smode_label, 0,
_N(smode_label) - 1, false); break;

#ifdef CW_DECODER
    case CWDEC:   paramAction(action, cwdec, 0x21, F("CW Decoder"), offon_label, 0, 1,
false); break;
#endif
#ifdef FILTER_700HZ
    case CWTONE:  if(dsp_cap) paramAction(action, cw_tone, 0x22, F("Tono CW"),
cw_tone_label, 0, 1, false); break;
#endif
#ifdef CW_LEARN
case LEARN_MODE: paramAction(action, learn_mode,  0x23, F("Learn Mode"),
learn_mode_label, 0, 2, false); break;
#endif
#ifdef SEMI_QSK
    case SEMIQSK: paramAction(action, semi_qsk,  0x24, F("Semi QSK"), offon_label, 0,
1, false); break;
#endif
#if defined(KEYER) || defined(CW_MESSAGE)
    case KEY_WPM:  paramAction(action, keyer_speed, 0x25, F("Keyer Speed"), NULL, 1,
60, false); break;
#endif
```

```
#ifdef KEYER
    case KEY_MODE: paramAction(action, keyer_mode,  0x26, F("Keyer Mode"),
keyer_mode_label, 0, 2, false); break;
    case KEY_PIN:  paramAction(action, keyer_swap,  0x27, F("Keyer Swap"), offon_label,
0, 1, false); break;
#endif
    case KEY_TX:   paramAction(action, practice,    0x28, F("Practica"), offon_label,
0, 1, false); break;
#ifdef VOX_ENABLE
    case VOX:     paramAction(action, vox,        0x31, F("VOX"), offon_label, 0, 1,
false); break;
    case VOXGAIN: paramAction(action, vox_thresh, 0x32, F("Nivel VOX"), NULL, 0, 255,
false); break;
#endif

#ifdef TX_DELAY
    case TXDELAY: paramAction(action, txdelay, 0x34, F("TX Delay"), NULL, 0, 255,
false); break;
#endif
#ifdef MOX_ENABLE
    case MOX:     paramAction(action, mox, 0x35, F("MOX"), NULL, 0, 2, false); break;
#endif
#ifdef CW_MESSAGE
    case CWINTERVAL: paramAction(action, cw_msg_interval, 0x41, F("Intervalo CQ"),
NULL, 0, 60, false); break;
    case CWMSG1:    paramAction(action, cw_msg[0],   0x42, F("Mensaje CQ  "),
sizeof(cw_msg)); break;
#ifdef CW_MESSAGE_EXT
    case CWMSG2:    paramAction(action, cw_msg[1],   0x43, F("Mensaje CW 1"),
sizeof(cw_msg)); break;
    case CWMSG3:    paramAction(action, cw_msg[2],   0x44, F("Mensaje CW 2"),
sizeof(cw_msg)); break;
    case CWMSG4:    paramAction(action, cw_msg[3],   0x45, F("Mensaje CW 3"),
sizeof(cw_msg)); break;
    case CWMSG5:    paramAction(action, cw_msg[4],   0x46, F("Mensaje CW 4"),
sizeof(cw_msg)); break;
    case CWMSG6:    paramAction(action, cw_msg[5],   0x47, F("Mensaje CW 5"),
sizeof(cw_msg)); break;
#endif
#endif
    case DRIVE:   paramAction(action, drive,   0x81, F("TX Drive"), NULL, 0, 8, false);
break;
    case PWM_MIN: paramAction(action, pwm_min, 0x82, F("PA Bias min"), NULL, 0, pwm_max
- 1, false); break;
    case PWM_MAX: paramAction(action, pwm_max, 0x83, F("PA Bias max"), NULL, pwm_min,
255, false); break;
    case SIFXTAL: paramAction(action, si5351.fxtal, 0x84, F("Ref freq"), NULL,
14000000, 28000000, false); break;
    case IQ_ADJ:  paramAction(action, rx_ph_q, 0x85, F("Fase I-Q"), NULL, 0, 180,
false); break;


    case BACKL:   paramAction(action, backlight, 0xA1, F("Backlight"), offon_label, 0,
1, false); break;   // workaround for varying N_PARAM and not being able to overflowing
default cases properly
    // Invisible parameters
    case FREQA:   paramAction(action, vfo[VFOA], 0, NULL, NULL, 0, 0, false); break;
    case FREQB:   paramAction(action, vfo[VFOB], 0, NULL, NULL, 0, 0, false); break;
    case MODEA:   paramAction(action, vfomode[VFOA], 0, NULL, NULL, 0, 0, false);
break;
    case MODEB:   paramAction(action, vfomode[VFOB], 0, NULL, NULL, 0, 0, false);
break;
    case VERS:    paramAction(action, eeprom_version, 0, NULL, NULL, 0, 0, false);
break;

    // Non-parameters
    case _NULL:   menumode = 0; show_banner(); change = true; break;
    default:      if((action == NEXT_MENU) && (id != N_PARAMS)) id =
paramAction(action, max(1 /*0*/, min(N_PARAMS, id + ((encoder_val > 0) ? 1 : -1))) );
break;  // keep iterating util menu item found
```

```
  }
  return id;
}

void initPins(){

  digitalWrite(SIG_OUT, LOW);
  digitalWrite(RX, HIGH);
  digitalWrite(KEY_OUT, LOW);
  digitalWrite(SIDETONE, LOW);
  pinMode(SIDETONE, OUTPUT);
  pinMode(SIG_OUT, OUTPUT);
  pinMode(RX, OUTPUT);
  pinMode(KEY_OUT, OUTPUT);
  pinMode(BUTTONS, INPUT);  // L/R/rotary button
  pinMode(DIT, INPUT_PULLUP);
  pinMode(DAH, INPUT);  // pull-up DAH 10k via AVCC
  //pinMode(DAH, INPUT_PULLUP); // Could this replace D4? But leaks noisy VCC into mic
input!
  digitalWrite(AUDIO1, LOW);  // when used as output, help can mute RX leakage into
AREF
  digitalWrite(AUDIO2, LOW);
  pinMode(AUDIO1, INPUT);
  pinMode(AUDIO2, INPUT);

#ifdef NTX
  digitalWrite(NTX, HIGH);
  pinMode(NTX, OUTPUT);
#endif //NTX
#ifdef PTX
  digitalWrite(PTX, LOW);
  pinMode(PTX, OUTPUT);
#endif //PTX


}

#ifdef CAT
// CAT support inspired by Charlie Morris, ZL2CTM, contribution by Alex, PE1EVX,
source: http://zl2ctm.blogspot.com/2020/06/digital-modes-transceiver.html?m=1
// https://www.kenwood.com/i/products/info/amateur/ts_480/pdf/ts_480_pc.pdf
#define CATCMD_SIZE   32
char CATcmd[CATCMD_SIZE];

void analyseCATcmd()
{
  if((CATcmd[0] == 'F') && (CATcmd[1] == 'A') && (CATcmd[2] == ';'))
    Command_GETFreqA();

  else if((CATcmd[0] == 'F') && (CATcmd[1] == 'A') && (CATcmd[13] == ';'))
    Command_SETFreqA();

  else if((CATcmd[0] == 'I') && (CATcmd[1] == 'F') && (CATcmd[2] == ';'))
    Command_IF();

  else if((CATcmd[0] == 'I') && (CATcmd[1] == 'D') && (CATcmd[2] == ';'))
    Command_ID();

  else if((CATcmd[0] == 'P') && (CATcmd[1] == 'S') && (CATcmd[2] == ';'))
    Command_PS();

  else if((CATcmd[0] == 'P') && (CATcmd[1] == 'S') && (CATcmd[2] == '1'))
    Command_PS1();

  else if((CATcmd[0] == 'A') && (CATcmd[1] == 'I') && (CATcmd[2] == ';'))
    Command_AI();

  else if((CATcmd[0] == 'A') && (CATcmd[1] == 'I') && (CATcmd[2] == '0'))
    Command_AI0();
```

```
  else if((CATcmd[0] == 'M') && (CATcmd[1] == 'D') && (CATcmd[2] == ';'))
    Command_GetMD();

  else if((CATcmd[0] == 'M') && (CATcmd[1] == 'D') && (CATcmd[3] == ';'))
    Command_SetMD();

  else if((CATcmd[0] == 'R') && (CATcmd[1] == 'X') && (CATcmd[2] == ';'))
    Command_RX();

  else if((CATcmd[0] == 'T') && (CATcmd[1] == 'X') && (CATcmd[2] == ';'))
    Command_TX0();

  else if((CATcmd[0] == 'T') && (CATcmd[1] == 'X') && (CATcmd[2] == '0'))
    Command_TX0();

  else if((CATcmd[0] == 'T') && (CATcmd[1] == 'X') && (CATcmd[2] == '1'))
    Command_TX1();

  else if((CATcmd[0] == 'T') && (CATcmd[1] == 'X') && (CATcmd[2] == '2'))
    Command_TX2();

  else if((CATcmd[0] == 'A') && (CATcmd[1] == 'G') && (CATcmd[2] == '0'))  // add
    Command_AG0();

  else if((CATcmd[0] == 'X') && (CATcmd[1] == 'T') && (CATcmd[2] == '1'))  // add
    Command_XT1();

  else if((CATcmd[0] == 'R') && (CATcmd[1] == 'T') && (CATcmd[2] == '1'))  // add
    Command_RT1();

  else if((CATcmd[0] == 'R') && (CATcmd[1] == 'C') && (CATcmd[2] == ';'))  // add
    Command_RC();

  else if((CATcmd[0] == 'F') && (CATcmd[1] == 'L') && (CATcmd[2] == '0'))  // need?
    Command_FL0();

  else if((CATcmd[0] == 'R') && (CATcmd[1] == 'S') && (CATcmd[2] == ';'))
    Command_RS();

  else if((CATcmd[0] == 'V') && (CATcmd[1] == 'X') && (CATcmd[2] != ';'))
    Command_VX(CATcmd[2]);

/*
The following CAT extensions are available to support remote operatons. Use a baudrate
of 115200 when enabling CAT_STREAMING config switch:

UA1;  enable audio streaming; an 8-bit audio stream nnn at 4812 samples/s is then sent
within CAT response USnnn; here nnn is of undefined length, it will end as soon other
CAT requests are processed and resume with a new USnnn; response
UA0;  disable audio streaming
UD;   requests display contents
UKnn; control keys, where nn is a sum of the following hexadecimal values: 0x80 encoder
left, 0x40 encoder right, 0x20 DIT/PTT, 0x10 DAH, 0x04 left-button, 0x02 encoder
button, 0x01 right button


The following Linux script may be useful to stream audio over CAT, play the audio and
redirect CAT to /tmp/ttyS0:

1. Pre-requisite installation: sudo apt-get install socat

2. Insert USB serial converter (/dev/ttyUSB0) and start:

stty -F /dev/ttyUSB0 raw -echo -echoe -echoctl -echoke 115200;

socat -d -d pty,link=/tmp/ttyS0,echo=0,ignoreeof,b115200,raw,perm=0777 pty,link=/tmp
/ttyS1,echo=0,ignoreeof,b115200,raw,perm=0777 &

sleep 5; cat /tmp/ttyS1 > /dev/ttyUSB0 &
```

```
cat /dev/ttyUSB0 | while IFS= read -n1 c; do
  case $state in
  0) if [ "$c" = ";" ]; then state=1; fi; printf "%s" "$c">>/tmp/ttyS1;
     ;;
  1) if [ "$c" = "U" ]; then state=2; else printf "%s" "$c">>/tmp/ttyS1; state=0; fi;
     ;;
  2) if [ "$c" = "S" ]; then state=3; else printf "U%s" "$c">>/tmp/ttyS1; state=0; fi;
     ;;
  3) if [ "$c" = ";" ]; then state=1; else printf "%s" "$c"; fi;
     ;;
  *) state=3;
     ;;
  esac
done | pacat --rate=7812 --channels=1 --format=u8 --latency-msec=30 --process-time-
msec=30 &

echo ";UA1;" > /tmp/ttyS0;

3. You should now hear audio, use pavucontrol. Now start your favorite digimode/logbook
application, and use /tmp/ttyS0 as CAT port.

4. Instead of step 3, you could visualize uSDX console:

clear; echo ";UA1;UD;" > /tmp/ttyS0; cat /tmp/ttyS0 | while IFS= read -d \; c; do echo
"${c}" |sed -E  's/^UD..(.+)$|^[^U][^D](.*)$/\1/g'| sed -E 's/^(.{16})(.+)$/\x1B[;1H
\x1B[1m\x1B[44m\x1B[97m\1\x1B[0m\x1B[K\n\x1B[1m\x1B[44m\x1B[97m\2\x1B[0m\x1B[K\n\x1B[K
\n\x1B[K/g'; echo ";UD;UD;" >> /tmp/ttyS0; sleep 1; done

Use pavumeter to set the correct mixer settings

*/
#ifdef CAT_EXT
  else if((CATcmd[0] == 'U') && (CATcmd[1] == 'K') && (CATcmd[4] == ';'))  // remote
key press
    Command_UK(CATcmd[2], CATcmd[3]);

  else if((CATcmd[0] == 'U') && (CATcmd[1] == 'D') && (CATcmd[2] == ';'))  // display
contents
    Command_UD();
#endif //CAT_EXT

#ifdef CAT_STREAMING
  else if((CATcmd[0] == 'U') && (CATcmd[1] == 'A') && (CATcmd[3] == ';'))  // audio
streaming enable/disable
    Command_UA(CATcmd[2]);
#endif //CAT_STREAMING

  else {
    Serial.print("?;");
  }
}

#ifdef CAT
volatile uint8_t cat_ptr = 0;
void serialEvent(){
  if(Serial.available()){
    rxend_event = millis() + 10;  // block display until this moment, to prevent CAT
cmds that initiate display changes to interfere with the next CAT cmd e.g. Hamlib:
FA00007071000;ID;
    char data = Serial.read();
    CATcmd[cat_ptr++] = data;
    if(data == ';'){
      CATcmd[cat_ptr] = '\0'; // terminate the array
      cat_ptr = 0;            // reset for next CAT command
#ifdef _SERIAL
      if(!cat_active){ cat_active = 1; smode = 0;} // disable smeter to reduce display
activity
#endif
#ifdef CAT_STREAMING
      if(cat_streaming){ noInterrupts(); cat_streaming = false; Serial.print(';'); }
```

```
// terminate CAT stream
      analyseCATcmd();   // process CAT cmd
      if(_cat_streaming){ Serial.print("US"); cat_streaming = true; }  // resume CAT
stream
      interrupts();
#else
      analyseCATcmd();
#endif // CAT_STREAMING
      delay(10);
    } else if(cat_ptr > (CATCMD_SIZE - 1)){ Serial.print("E;"); cat_ptr = 0; } //
overrun
  }
}
#endif //CAT

#ifdef CAT_EXT
void Command_UK(char k1, char k2)
{
  cat_key = ((k1 - '0') << 4) | (k2 - '0');
  if(cat_key & 0x40){ encoder_val--; cat_key &= 0x3f; }
  if(cat_key & 0x80){ encoder_val++; cat_key &= 0x3f; }
  char Catbuffer[16];
  sprintf(Catbuffer, "UK%c%c;", k1, k2);
  Serial.print(Catbuffer);
}

void Command_UD()
{
  char Catbuffer[40];
  sprintf(Catbuffer, "UD%02u%s;", (lcd.curs) ? lcd.y*16+lcd.x : 16*2+1, lcd.text);
  Serial.print(Catbuffer);
}

void Command_UA(char en)
{
  char Catbuffer[16];
  sprintf(Catbuffer, "UA%01u;", (_cat_streaming = (en == '1')));
  Serial.print(Catbuffer);
  if(_cat_streaming){ Serial.print("US"); cat_streaming = true; }
}
#endif // CAT_EXT

void Command_GETFreqA()
{
#ifdef _SERIAL
  if(!cat_active) return;
#endif
  char Catbuffer[32];
  unsigned int g,m,k,h;
  uint32_t tf;

  tf=freq;
  g=(unsigned int)(tf/1000000000lu);
  tf-=g*1000000000lu;
  m=(unsigned int)(tf/1000000lu);
  tf-=m*1000000lu;
  k=(unsigned int)(tf/1000lu);
  tf-=k*1000lu;
  h=(unsigned int)tf;

  sprintf(Catbuffer,"FA%02u%03u",g,m);
  Serial.print(Catbuffer);
  sprintf(Catbuffer,"%03u%03u;",k,h);
  Serial.print(Catbuffer);
}

void Command_SETFreqA()
{
  char Catbuffer[16];
  strncpy(Catbuffer,CATcmd+2,11);
```

```
  Catbuffer[11]='\0';

  freq=(uint32_t)atol(Catbuffer);
  change=true;
}

void Command_IF()
{
#ifdef _SERIAL
  if(!cat_active) return;
#endif
  char Catbuffer[32];
  unsigned int g,m,k,h;
  uint32_t tf;

  tf=freq;
  g=(unsigned int)(tf/1000000000lu);
  tf-=g*1000000000lu;
  m=(unsigned int)(tf/1000000lu);
  tf-=m*1000000lu;
  k=(unsigned int)(tf/1000lu);
  tf-=k*1000lu;
  h=(unsigned int)tf;

  sprintf(Catbuffer,"IF%02u%03u%03u%03u",g,m,k,h);
  Serial.print(Catbuffer);
  sprintf(Catbuffer,"00000+000000");
  Serial.print(Catbuffer);
  sprintf(Catbuffer,"0000");
  Serial.print(Catbuffer);
  Serial.print(mode + 1);
  sprintf(Catbuffer,"0000000;");
  Serial.print(Catbuffer);
}

void Command_AI()
{
  Serial.print("AI0;");
}

void Command_AG0()
{
  Serial.print("AG0;");
}

void Command_XT1()
{
  Serial.print("XT1;");
}

void Command_RT1()
{
  Serial.print("RT1;");
}

void Command_RC()
{
  rit = 0;
  Serial.print("RC;");
}

void Command_FL0()
{
  Serial.print("FL0;");
}

void Command_GetMD()
{
  Serial.print("MD");
  Serial.print(mode + 1);
```

```
  Serial.print(';');
}

void Command_SetMD()
{
  mode = CATcmd[2] - '1';

  vfomode[vfosel%2] = mode;
  change = true;
  si5351.iqmsa = 0;  // enforce PLL reset
}

void Command_AI0()
{
  Serial.print("AI0;");
}

void Command_RX()
{

  switch_rxtx(0);
  semi_qsk_timeout = 0;  // hack: fix for multiple RX cmds

  Serial.print("RX0;");
}

void Command_TX0()
{

  switch_rxtx(1);

}

void Command_TX1()
{

  switch_rxtx(1);

}

void Command_TX2()
{

  switch_rxtx(1);

}

void Command_RS()
{
  Serial.print("RS0;");
}

void Command_VX(char mode)
{
  char Catbuffer[16];
  sprintf(Catbuffer, "VX%c;",mode);
  Serial.print(Catbuffer);
}

void Command_ID()
{
  Serial.print("ID020;");
}

void Command_PS()
{
  Serial.print("PS1;");
}

void Command_PS1()
```

```
{
}
#endif //CAT

void fatal(const __FlashStringHelper* msg, int value = 0, char unit = '\0') {
  lcd.setCursor(0, 1);
  lcd.print('!'); lcd.print('!');
  lcd.print(msg);
  if(unit != '\0') {
    lcd.print('=');
    lcd.print(value);
    lcd.print(unit);
  }
  lcd_blanks();
  delay(1500);
  wdt_reset();
}

//refresh LUT based on pwm_min, pwm_max
void build_lut()
{
  for(uint16_t i = 0; i != 256; i++)    // refresh LUT based on pwm_min, pwm_max
    lut[i] = (i * (pwm_max - pwm_min)) / 255 + pwm_min;
    //lut[i] = min(pwm_max, (float)106*log(i) + pwm_min);  // compressed microphone
output: drive=0, pwm_min=115, pwm_max=220
}


#ifdef CW_LEARN
void practice_cw()
{
     if (loop1){                          //genera nuevo indicativo
        randomSeed(millis());
        cw_to_send_to_user = (char)random(65,91);
        cw_to_send_to_user.concat((char)random(65,91));
        cw_to_send_to_user.concat((char)random(48,58)); // numeros del 0 al 9
        cw_to_send_to_user.concat((char)random(65,91));
        cw_to_send_to_user.concat((char)random(65,91));
        cw_to_send_to_user.concat((char)random(65,91));
        loop1=0;
        }

      if (letra == '?')kn = 0;  //repite el indicativo en cualquier momento

        for (kn; kn <=6 ;kn++){
        cw_tx(cw_to_send_to_user[kn]);
        lcd.setCursor(kn,0);
        lcd.print(cw_to_send_to_user[kn]);
        gx=0;
        letra="";
        }

      if (letra == cw_to_send_to_user[gx]){     //compara cadena generada y recibida
             lcd.setCursor(7+gx,0);
             lcd.print(letra);
             gx++;
             if (gx == 6){ loop1=1; kn=0; lcd.setCursor(0,0);lcd.print("
");
             }
             }
 }


/*
void learn_cw_koch()
{
  const static char koch[] =
{'K','M','R','S','U','A','P','T','L','O','W','I','.','N','J','E','F',

'0','Y','V',',','G','5','/','Q','9','Z','H','3','8','B','?','4','2','7','C','1','D','6'
```

```
,'X','\0'};
  char cw_txl[17];   // Buffer letra enviada
  byte i;
  byte j;
  boolean error = false;



  randomSeed(micros()); // random seed = microseconds since start.
//        Por si quiero usar aleatorios en codigo ascii
//        cw_to_send_to_user = ((char)random(65,91)); codigo ascii de la A a la Z
//        cw_to_send_to_user = ((char)random(48,58)); codigo ascii del 0 al 9


//      } while (rx_cnt < GROUP_NUM && !error); podriamos salir si
!digitalRead(BUTTONS) break

      if (error) { lcd.print("  MAL");
      }
      else { lcd.print("  BIEN");
      }

}    */
#endif

void setup()
{
  digitalWrite(KEY_OUT, LOW);  // for safety: to prevent exploding PA MOSFETs, in case
there was something still biasing them.
  si5351.powerDown();  // disable all CLK outputs (especially needed for si5351
variants that has CLK2 enabled by default, such as Si5351A-B04486-GT)

  uint8_t mcusr = MCUSR;
  MCUSR = 0;
  //wdt_disable();
  wdt_enable(WDTO_4S);  // Enable watchdog
  uint32_t t0, t1;

  ADMUX = (1 << REFS0);  // restore reference voltage AREF (5V)

  // disable external interrupts
  PCICR = 0;
  PCMSK0 = 0;
  PCMSK1 = 0;
  PCMSK2 = 0;

  encoder_setup();

  initPins();

  delay(100);           // at least 40ms after power rises above 2.7V before sending
commands
  lcd.begin(16, 2);  // Init LCD

  for(i = 0; i != N_FONTS; i++){  // Init fonts
    pgm_cache_item(fonts[i], 8);
    lcd.createChar(0x01 + i, /*fonts[i]*/_item);
  }


  show_banner();
  lcd.setCursor(5, 0); lcd.print("EA2EHC"); lcd_blanks();
  lcd.setCursor(5, 1); lcd.print(F(VERSION)); lcd_blanks();
  delay(1500);
  drive = 4;  // Init settings
  cw_offset = tones[cw_tone];
  //freq = bands[band];

  // Load parameters from EEPROM, reset to factory defaults when stored values are from
a different version
```

```
  paramAction(LOAD, VERS);
  if((eeprom_version != get_version_id()) || _digitalRead(BUTTONS) ){  // EEPROM clean:
if rotary-key pressed or version signature in EEPROM does NOT corresponds with this
firmware
    eeprom_version = get_version_id();
    //for(int n = 0; n != 1024; n++){ eeprom_write_byte((uint8_t *) n, 0); wdt_reset();
} //clean EEPROM
    //eeprom_write_dword((uint32_t *)EEPROM_OFFSET/3, 0x000000);
    paramAction(SAVE);  // save default parameter values
    lcd.setCursor(0, 1); lcd.print(F("Reset settings.."));
    delay(500); wdt_reset();
  } else {
    paramAction(LOAD);  // load all parameters
  }
  si5351.iqmsa = 0;  // enforce PLL reset
  change = true;
  prev_bandval = bandval;
  vox = false;  // disable VOX
  nr = 0; // disable NR
  rit = false;  // disable RIT
  freq = vfo[vfosel%2];
  mode = vfomode[vfosel%2];
  build_lut();
  show_banner();  // remove release number
  start_rx();

#if defined CAT
#ifdef CAT_STREAMING
  #define BAUD   115200           //Baudrate used for serial communications
#else
  #define BAUD   38400            //38400 //115200 //4800 //Baudrate used for serial
communications (CAT)
#endif
  Serial.begin(16000000ULL * BAUD / F_MCU); // corrected for F_CPU=20M
  Command_IF();

#endif //CAT

#ifdef KEYER
  keyerState = IDLE;
  keyerControl = IAMBICB;      // Or 0 for IAMBICA
  loadWPM(keyer_speed);        // Fix speed at 15 WPM
#endif //KEYER

  for(; !_digitalRead(DIT) || ((mode == CW) && (!_digitalRead(DAH)));){ // wait until
DIH/DAH/PTT is released to prevent TX on startup
    lcd.setCursor(15, 1); lcd.print('T');
    delay(1000);
    lcd.setCursor(15, 1); lcd.print(' ');
    delay(1000);
    wdt_reset();
  }
}

static int32_t _step = 0;

void loop()
{
#ifdef VOX_ENABLE
  if((vox) && ((mode == LSB) || (mode == USB))){  // If VOX enabled (and in LSB/USB
mode), then take mic samples and feed ssb processing function, to derive amplitude, and
potentially detect cross vox_threshold to detect a TX or RX event: this is expressed in
tx variable
    if(!vox_tx){ // VOX not active
#ifdef MULTI_ADC
      if(vox_sample++ == 16){  // take N sample, then process
        ssb(((int16_t)(vox_adc/16) - (512 - AF_BIAS)) >> MIC_ATTEN);   // sampling mic
        vox_sample = 0;
        vox_adc = 0;
      } else {
```

```
        vox_adc += analogSampleMic();
      }
#else
      ssb(((int16_t)(analogSampleMic()) - 512) >> MIC_ATTEN);   // sampling mic
#endif
      if(tx){  // TX triggered by audio -> TX
        vox_tx = 1;
        switch_rxtx(255);
        //for(;(tx);) wdt_reset();  // while in tx (workaround for RFI feedback related
issue)
        //delay(100); tx = 255;
      }
    } else if(!tx){  // VOX activated, no audio detected -> RX
      switch_rxtx(0);
      vox_tx = 0;
      delay(32); //delay(10);
      //vox_adc = 0; for(i = 0; i != 32; i++) ssb(0); //clean buffers
      //for(int i = 0; i != 32; i++) ssb((analogSampleMic() - 512) >> MIC_ATTEN); //
clear internal buffer
      //tx = 0; // make sure tx is off (could have been triggered by rubbish in above
statement)
    }
  }
#endif //VOX_ENABLE

#ifdef CW_DECODER
  //if((mode == CW) && cwdec) cw_decode();  // if(!(semi_qsk_timeout)) cw_decode();
else dec2();
  //if((mode == CW) && cwdec && ((!tx) && (!semi_qsk_timeout))) cw_decode();  // CW
decoder only active during RX
  if((mode == CW) && cwdec && ((!tx) && (!semi_qsk_timeout))) cw_decode();  // CW
decoder only active during RX
#endif  //CW_DECODER

  if(menumode == 0){ // in main
#ifdef CW_DECODER
    if(cw_event){
      uint8_t offset = (uint8_t[]){ 0, 7, 3, 5, 3, 7, 8 }[smode]; // depending on
smeter more/less cw-text
      lcd.noCursor();

      cw_event = false;
      lcd.setCursor(0, 0); lcd.print(out + offset);

      stepsize_showcursor();
    }

#endif  //CW_DECODER
      if((!semi_qsk_timeout) && (!vox_tx))
        smeter();
  }

#ifdef KEYER  //Keyer
  if(mode == CW && keyer_mode != SINGLE){  // check DIT/DAH keys for CW

    switch(keyerState){ // Basic Iambic Keyer, keyerControl contains processing flags
and keyer mode bits, Supports Iambic A and B, State machine based, uses calls to
millis() for timing.
    case IDLE: // Wait for direct or latched paddle press
        if((_digitalRead(DAH) == LOW) ||
            (_digitalRead(DIT) == LOW) ||
            (keyerControl & 0x03))
        {
#ifdef CW_MESSAGE
            cw_msg_event = 0;  // clear cw message event
#endif //CW_MESSAGE
            update_PaddleLatch();
            keyerState = CHK_DIT;
        }
        break;
```

```
    case CHK_DIT: // See if the dit paddle was pressed
        if(keyerControl & DIT_L) {
            keyerControl |= DIT_PROC;
            ktimer = ditTime;
            keyerState = KEYED_PREP;
        } else {
            keyerState = CHK_DAH;
        }
        break;
    case CHK_DAH: // See if dah paddle was pressed
        if(keyerControl & DAH_L) {
            ktimer = ditTime*3;
            keyerState = KEYED_PREP;
        } else {
            keyerState = IDLE;
        }
        break;
    case KEYED_PREP: // Assert key down, start timing, state shared for dit or dah
        Key_state = HIGH;
        switch_rxtx(Key_state);
        ktimer += millis();                 // set ktimer to interval end time
        keyerControl &= ~(DIT_L + DAH_L);   // clear both paddle latch bits
        keyerState = KEYED;                 // next state
        break;
    case KEYED: // Wait for timer to expire
        if(millis() > ktimer) {            // are we at end of key down ?
            Key_state = LOW;
            switch_rxtx(Key_state);
            ktimer = millis() + ditTime;    // inter-element time
            keyerState = INTER_ELEMENT;     // next state
        } else if(keyerControl & IAMBICB) {
            update_PaddleLatch();           // early paddle latch in Iambic B mode
        }
        break;
    case INTER_ELEMENT:
        // Insert time between dits/dahs
        update_PaddleLatch();               // latch paddle state
        if(millis() > ktimer) {            // are we at end of inter-space ?
            if(keyerControl & DIT_PROC) {             // was it a dit or dah ?
                keyerControl &= ~(DIT_L + DIT_PROC);   // clear two bits
                keyerState = CHK_DAH;                  // dit done, check for dah
            } else {
                keyerControl &= ~(DAH_L);              // clear dah latch
                keyerState = IDLE;                     // go idle
            }
        }
        break;
    }

  } else {
#endif //KEYER


  uint8_t pin = ((mode == CW) && (keyer_swap)) ? DAH : DIT;
  if(!vox_tx)  //  ONLY if VOX not active, then check DIT/DAH (fix for VOX to prevent
RFI feedback through EMI on DIT or DAH line)
   if(!_digitalRead(pin)){  // PTT/DIT keys transmitter
#ifdef CW_MESSAGE
    cw_msg_event = 0;  // clear cw message event
#endif //CW_MESSAGE
    switch_rxtx(1);
    do {
      wdt_reset();
      delay((mode == CW) ? 10 : 100);  // keep the tx keyed for a while before sensing
(helps against RFI issues on DAH/DAH line)

      if(_digitalRead(BUTTONS)) break;  // break if button is pressed (to prevent
potential lock-up)
    } while(!_digitalRead(pin)); // until released
    switch_rxtx(0);
```

```
   }

#ifdef KEYER
  }
#endif //KEYER

#ifdef SEMI_QSK
  if((semi_qsk_timeout) && (millis() > semi_qsk_timeout)){ switch_rxtx(0); }  //
delayed QSK RX
#endif
  enum event_t { BL=0x10, BR=0x20, BE=0x30, SC=0x01, DC=0x02, PL=0x04, PLC=0x05,
PT=0x0C }; // button-left, button-right and button-encoder; single-click, double-click,
push-long, push-and-turn
  if(_digitalRead(BUTTONS)){   // Left-/Right-/Rotary-button (while not already
pressed)
    if(!((event & PL) || (event & PLC))){  // hack: if there was long-push before, then
fast forward
      uint16_t v = analogSafeRead(BUTTONS);
#ifdef CAT_EXT
      if(cat_key){ v = (cat_key&0x04) ? 512 : (cat_key&0x01) ? 870 : (cat_key&0x02) ?
1024 : 0; }  // override analog value exercised by BUTTONS press
#endif //CAT_EXT
      event = SC;
      int32_t t0 = millis();
      for(; _digitalRead(BUTTONS);){ // until released or long-press
        if((millis() - t0) > 300){ event = PL; break; }
        wdt_reset();
      }
      delay(10); //debounce
      for(; (event != PL) && ((millis() - t0) < 500);){ // until 2nd press or timeout
        if(_digitalRead(BUTTONS)){ event = DC; break; }
        wdt_reset();
      }
      for(; _digitalRead(BUTTONS);){ // until released, or encoder is turned while
longpress
        if(encoder_val && event == PL){ event = PT; break; }
        wdt_reset();
      }  // Max. voltages at ADC3 for buttons L,R,E: 3.76V;4.55V;5V, thresholds are in
center
      event |= (v < (4.2 * 1024.0 / 5.0)) ? BL : (v < (4.8 * 1024.0 / 5.0)) ? BR : BE;
// determine which button pressed based on threshold levels
    } else {  // hack: fast forward handling
      event = (event&0xf0) | ((encoder_val) ? PT : PLC/*PL*/);  // only alternate
between push-long/turn when applicable
    }
    switch(event){
      case BL|PL:  // Called when menu button pressed
      case BL|PLC: // or kept pressed
        menumode = 2;
        break;
      case BL|PT:
        menumode = 1;
        //if(menu == 0) menu = 1;
        break;
      case BL|SC:
#ifdef CW_MESSAGE
        if((menumode == 1) && (menu >= CWMSG1) && (menu <= CWMSG6)){
          cw_msg_event = millis();
          cw_msg_id = menu - CWMSG1;
          menumode = 0;
          break;
        }
#endif //CW_MESSAGE
        int8_t _menumode;
        if(menumode == 0){ _menumode = 1; if(menu == 0) menu = 1; }  // short left-
click while in default screen: enter menu mode
        if(menumode == 1){ _menumode = 2; }                          // short left-
click while in menu: enter value selection screen
        if(menumode >= 2){ _menumode = 0; paramAction(SAVE, menu); } // short left-
click while in value selection screen: save, and return to default screen
```

```
        menumode = _menumode;
        break;
      case BL|DC:
        break;
      case BR|SC:
        if(!menumode){
          int8_t prev_mode = mode;
          if(rit){ rit = 0; stepsize = prev_stepsize[mode == CW]; change = true;
break; }
          mode += 1;
          //encoder_val = 1;
          //paramAction(UPDATE, MODE); // Mode param //paramAction(UPDATE, mode, NULL,
F("Mode"), mode_label, 0, _N(mode_label), true);
//#define MODE_CHANGE_RESETS  1
#ifdef MODE_CHANGE_RESETS
          if(mode != CW) stepsize = STEP_1k; else stepsize = STEP_500; // sets suitable
stepsize
#endif
          if(mode > CW) mode = LSB;  // skip all other modes (only LSB, USB, CW)
#ifdef MODE_CHANGE_RESETS
          if(mode == CW) { filt = 4; nr  = 0; } else filt = 0;  // resets filter (to
most BW) and NR on mode change
#else
          if(mode == CW) { nr  = 0; }
          prev_stepsize[prev_mode == CW] = stepsize; stepsize = prev_stepsize[mode ==
CW]; // backup stepsize setting for previous mode, restore previous stepsize setting
for current selected mode; filter settings captured for either CQ or other modes.
          prev_filt[prev_mode == CW] = filt; filt = prev_filt[mode == CW];  // backup
filter setting for previous mode, restore previous filter setting for current selected
mode; filter settings captured for either CQ or other modes.
#endif
          //paramAction(UPDATE, MODE);
          vfomode[vfosel%2] = mode;
          paramAction(SAVE, (vfosel%2) ? MODEB : MODEA);  // save vfoa/b changes
          paramAction(SAVE, MODE);
          paramAction(SAVE, FILTER);
          si5351.iqmsa = 0;  // enforce PLL reset
#ifdef CW_DECODER
          if((prev_mode == CW) && (cwdec)) show_banner();
#endif
          change = true;
        } else {
          if(menumode == 1){ menumode = 0; }  // short right-click while in menu: enter
value selection screen
          if(menumode >= 2){ menumode = 1; change = true; paramAction(SAVE, menu); } //
short right-click while in value selection screen: save, and return to menu screen
        }
        break;
      case BR|DC:
        filt++;
        _init = true;
        if(mode == CW && filt > N_FILT) filt = 4;
        if(mode == CW && filt == 4) stepsize = STEP_500; // reset stepsize for 500Hz
filter
        if(mode == CW && (filt == 5 || filt == 6) && stepsize < STEP_100) stepsize =
STEP_100; // for CW BW 200, 100      -> step = 100 Hz
        if(mode == CW && filt == 7 && stepsize < STEP_10) stepsize = STEP_10;
// for CW BW 50 -> step = 10 Hz
        if(mode != CW && filt > 3) filt = 0;
        encoder_val = 0;
        paramAction(UPDATE, FILTER);
        paramAction(SAVE, FILTER);
        wdt_reset(); delay(1500); wdt_reset();
        change = true; // refresh display
        break;
      case BR|PL:
#ifdef SIMPLE_RX
        // Experiment: ISR-less sdr_rx():
        smode = 0;
        TIMSK2 &= ~(1 << OCIE2A);  // disable timer compare interrupt
```

```
        delay(100);
        lcd.setCursor(15, 1); lcd.print('X');
        static uint8_t x = 0;
        uint32_t next = 0;
        for(;;){
          func_ptr();
          if(!rx_state){
            x++;
            if(x > 16){
              loop();
              //lcd.setCursor(9, 0); lcd.print((int16_t)100); lcd.print(F("dBm   "));
// delays are taking too long!
              x= 0;
            }
          }
          //for(;micros() < next;);  next = micros() + 16;   // sync every
1000000/62500=16ms (or later if missed)
        } //
#endif //SIMPLE_RX
#ifdef RIT_ENABLE
        rit = !rit;
        stepsize = (rit) ?  STEP_10 : prev_stepsize[mode == CW];
        if(!rit){  // after RIT comes VFO A/B swap
          vfosel = !vfosel;
          freq = vfo[vfosel%2];  // todo: share code with menumode
          mode = vfomode[vfosel%2];
          // make more generic:
          if(mode != CW) stepsize = STEP_1k; else stepsize = STEP_500;
          if(mode == CW) { filt = 4; nr = 0; } else filt = 0;
        }
        change = true;
#endif //RIT_ENABLE
        break;

#ifdef TUNING_DIAL
      case BR|PLC:  // while pressed long continues
      case BE|PLC:
        freq = freq + ((_step > 0) ? 1 : -1) * pow(2, abs(_step)); change=true;
        break;
      case BR|PT:
        _step += encoder_val; encoder_val = 0;
        lcd.setCursor(0, 0); lcd.print(_step); lcd_blanks();
        break;
#endif //TUNING_DIAL
      case BE|SC:
        if(!menumode){
          stepsize_change(+1);
        } else {
          int8_t _menumode;
          if(menumode == 1){ _menumode = 2; }  // short encoder-click while in menu:
enter value selection screen
          if(menumode == 2){ _menumode = 1; change = true; paramAction(SAVE, menu); }
// short encoder-click while in value selection screen: save, and return to menu screen
#ifdef MENU_STR
          if(menumode == 3){ _menumode = 3; paramAction(NEXT_CH, menu); } // short
encoder-click while in string edit mode: change position to next character
#endif
          menumode = _menumode;
        }
        break;
      case BE|DC:
        //delay(100);
        bandval++;
        //if(bandval >= N_BANDS) bandval = 0;
        if(bandval >= (N_BANDS-1)) bandval = 1;  // excludes 6m, 160m
        stepsize = STEP_1k;
        change = true;
        break;
      case BE|PL: stepsize_change(-1); break;
      case BE|PT:
```

```
          for(; _digitalRead(BUTTONS);){ // process encoder changes until released
          wdt_reset();
          if(encoder_val){
            paramAction(UPDATE, VOLUME);
            if(volume < 0){ volume = 10; paramAction(SAVE, VOLUME); powerDown(); }  //
powerDown when volume < 0
            paramAction(SAVE, VOLUME);
          }
        }
        change = true; // refresh display
        break;

    }
  } else event = 0;  // no button pressed: reset event

  if((menumode) || (prev_menumode != menumode)){  // Show parameter and value
    int8_t encoder_change = encoder_val;
    if((menumode == 1) && encoder_change){
      menu += encoder_val;   // Navigate through menu

      menu = max(1 /* 0 */, min(menu, N_PARAMS));

      menu = paramAction(NEXT_MENU, menu);  // auto probe next menu item (gaps may
exist)
      encoder_val = 0;
    }
    if(encoder_change || (prev_menumode != menumode)) paramAction(UPDATE_MENU,
(menumode) ? menu : 0);  // update param with encoder change and display
    prev_menumode = menumode;
    if(menumode == 2){
      if(encoder_change){
        lcd.setCursor(0, 1); lcd.cursor();  // edits menu item value; make cursor
visible
        if(menu == MODE){ // post-handling Mode parameter
          vfomode[vfosel%2] = mode;
          paramAction(SAVE, (vfosel%2) ? MODEB : MODEA);  // save vfoa/b changes
          change = true;
          si5351.iqmsa = 0;  // enforce PLL reset
          // make more generic:
          if(mode != CW) stepsize = STEP_1k; else stepsize = STEP_500;
          if(mode == CW) { filt = 4; nr = 0; } else filt = 0;
        }
        if(menu == BAND){
          change = true;
        }
        //if(menu == NR){ if(mode == CW) nr = false; }
        if(menu == VFOSEL){
          freq = vfo[vfosel%2];
          mode = vfomode[vfosel%2];
          // make more generic:
          if(mode != CW) stepsize = STEP_1k; else stepsize = STEP_500;
          if(mode == CW) { filt = 4; nr = 0; } else filt = 0;
          change = true;
        }
#ifdef RIT_ENABLE
        if(menu == RIT){
          stepsize = (rit) ?  STEP_10 : STEP_500;
          change = true;
        }
#endif
        //if(menu == VOX){ if(vox){ vox_thresh-=1; } else { vox_thresh+=1; }; }
        if(menu == ATT){ // post-handling ATT parameter
          if(dsp_cap == SDR){
            noInterrupts();
          #ifdef SWAP_RX_IQ
            adc_start(1, !(att & 0x01)/*true*/, F_ADC_CONV); admux[0] = ADMUX;
            adc_start(0, !(att & 0x01)/*true*/, F_ADC_CONV); admux[1] = ADMUX;
          #else
            adc_start(0, !(att & 0x01)/*true*/, F_ADC_CONV); admux[0] = ADMUX;
            adc_start(1, !(att & 0x01)/*true*/, F_ADC_CONV); admux[1] = ADMUX;
```

```
          #endif //SWAP_RX_IQ
            interrupts();
          }
          digitalWrite(RX, !(att & 0x02)); // att bit 1 ON: attenuate -20dB by
disabling RX line, switching Q5 (antenna input switch) into 100k resistence
          pinMode(AUDIO1, (att & 0x04) ? OUTPUT : INPUT); // att bit 2 ON: attenuate
-40dB by terminating ADC inputs with 10R
          pinMode(AUDIO2, (att & 0x04) ? OUTPUT : INPUT);
        }
        if(menu == SIFXTAL){
          change = true;
        }

        if((menu == PWM_MIN) || (menu == PWM_MAX)){
          build_lut();
        }

        if(menu == CWTONE){
          if(dsp_cap){ cw_offset = (cw_tone == 0) ? tones[0] : tones[1];
paramAction(SAVE, CWOFF); }
        }
        if(menu == IQ_ADJ){
          change = true;
        }

#ifdef KEYER
        if(menu == KEY_WPM){
          loadWPM(keyer_speed);
        }
        if(menu == KEY_MODE){
          if(keyer_mode == 0){ keyerControl = IAMBICA; }
          if(keyer_mode == 1){ keyerControl = IAMBICB; }
          if(keyer_mode == 2){ keyerControl = SINGLE; }
        }
#endif //KEYER
#ifdef TX_DELAY
        if(menu == TXDELAY){
          semi_qsk = (txdelay > 0);
        }
#endif //TX_DELAY
      }
    }
  }

  if(menumode == 0){
    if(encoder_val){  // process encoder tuning steps
      process_encoder_tuning_step(encoder_val);
      encoder_val = 0;
    }
  }

  if((change) && (!tx) && (!vox_tx)){  // only change if TX is OFF, prevent
simultaneous I2C bus access
    change = false;
    if(prev_bandval != bandval){ freq = band[bandval]; prev_bandval = bandval; }
    vfo[vfosel%2] = freq;
    save_event_time = millis() + 1000;  // schedule time to save freq (no save while
tuning, hence no EEPROM wear out)

    if(menumode == 0){
      display_vfo(freq);
      stepsize_showcursor();
#ifdef CAT
      //Command_GETFreqA();
#endif

      // The following is a hack for SWR measurement:
      //si5351.alt_clk2(freq + 2400);
      //si5351.SendRegister(SI_CLK_OE, 0b11111000); // CLK2_EN=1, CLK1_EN,CLK0_EN=1
      //digitalWrite(SIG_OUT, HIGH);  // inject CLK2 on antenna input via 120K
```

```
    }

    //noInterrupts();
    if(mode == CW){
      si5351.freq(freq + cw_offset, rx_ph_q, 0/*90, 0*/);  // RX in CW-R (=LSB),
correct for CW-tone offset
    } else
    if(mode == LSB)
      si5351.freq(freq, rx_ph_q, 0/*90, 0*/);  // RX in LSB
    else
      si5351.freq(freq, 0, rx_ph_q/*0, 90*/);  // RX in USB, ...
#ifdef RIT_ENABLE
    if(rit){ si5351.freq_calc_fast(rit); si5351.SendPLLRegisterBulk(); }
#endif //RIT_ENABLE
    //interrupts();
  }

  if((save_event_time) && (millis() > save_event_time)){  // save freq when time has
reached schedule
    paramAction(SAVE, (vfosel%2) ? FREQB : FREQA);  // save vfoa/b changes
    save_event_time = 0;
    //lcd.setCursor(15, 1); lcd.print('S'); delay(100); lcd.setCursor(15, 1);
lcd.print('R');
  }

#ifdef CW_MESSAGE
  if((mode == CW) && (cw_msg_event) && (millis() > cw_msg_event)){  // if it is time to
send a CW message
    if((cw_tx(cw_msg[cw_msg_id]) == 0) && ((cw_msg[cw_msg_id][0] == 'C') &&
(cw_msg[cw_msg_id][1] == 'Q')) && cw_msg_interval) cw_msg_event = millis() + 1000 *
cw_msg_interval; else cw_msg_event = 0;  // then send message, if not interrupted and
its a CQ msg and there is an interval set, then schedule new event
  }
#endif //CW_MESSAGE


#ifdef CW_LEARN


   if(learn_mode == 1){
//      lcd.setCursor(0, 0);  Aqui tendremos que salir a la pantalla principal, poner
en modo practica para no tx, y cambiar el modo a cw
//      lcd.print(learn_mode);

//      digitalWrite(RX, LOW); // TX (disable RX)
//      lcd.setCursor(15, 1); lcd.print('P');
//     si5351.SendRegister(SI_CLK_OE, 0b11111111); // CLK2_EN,CLK1_EN,CLK0_EN=0


//if(learn_mode == 1)
        loop1=1;
        practice_cw();

//if(learn_mode == 2)

//       learn_cw_koch()

       }
#endif
  wdt_reset();

  //{ lcd.setCursor(0, 0); lcd.print(freeMemory()); lcd.print(F("    ")); }
}
```