```
////////////////////////////////////////////////////////////////////////
// Harris RF550 / Teletron TE704C remote VFO & PC interface.          //
// Connects to the RF550/TE704C remote frequency control connector.   //
//                                                                    //
// Copyright (C) 2018, 2026 Giuseppe Perotti, I1EPJ, i1epj@aricasale.it //
//                                                                    //
// Version 1.1                                                        //
////////////////////////////////////////////////////////////////////////
//                                                                    //
// This program is free software: you can redistribute it and/or modify //
// it under the terms of the GNU General Public License as published by //
// the Free Software Foundation, either version 3 of the License, or  //
// (at your option) any later version.                                //
//                                                                    //
// This program 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 General Public License for more details.                       //
//                                                                    //
// You should have received a copy of the GNU General Public License  //
// along with this program.  If not, see <http://www.gnu.org/licenses/>. //
//                                                                    //
//
////////////////////////////////////////////////////////////////////////

//
// ------------------------- Receiver configuration. -------------------------
//
// Uncomment ONLY ONE of the following.
// #define TE704C // Not tested since no hardware is currently available!
#define RF550

// Uncomment this if you use inverting buffers (e.g. 7405).
// Default is to use non-inverting buffers (e.g. 7407 or 4504).
// #define NOT_BUFFERS

// Comment this if you don't want the last VFO frequency/memory to be
// automatically saved in EEPROM. Saving happens once after 1min of inactivity
// (no frequency or memory change) and only if the current frequency/memory
// has changed in the meantime, or when the REMOTE line goes low.
// Undefining the following option disables only the timed save, the saving
// when the REMOTE line goes low is always active.
#define SAVE_LAST_FREQMEM

// Uncomment this if you want the T command
// (bit test, for testing purposes only).
// #define BIT_TEST

//
// -------------. Configuration end, i.e. hic sunt leones ---------------------
//

#include <Encoder.h>
#include <LiquidCrystal_PCF8574.h>
#include <EEPROM.h>

// Arduino MEGA pins.
// Output pins are defined in bitaddr[] and are initializad from there.
#define ENCODER_CLK 2  // Encoder CLOCK.
#define ENCODER_DT 3   // Encoder DATA.
#define ENCODER_SW 4   // Encoder SWITCH.
#ifdef RF550
#define REMOTE 46  // RX remote-enabled input.
#endif
```

```
#ifdef TE704C
#define REMOTE 48
#endif

// Times in milliseconds for short, medium amd long encoder switch press.
// Any time > LPRESS_TIME is recognized as a very long press.
#define SPRESS_TIME 1000
#define MPRESS_TIME 2000
#define LPRESS_TIME 4000

// Frequency coverage.
#ifdef TE704C
#define MIN_FREQ 10000
#define MAX_FREQ 30000000
#endif
#ifdef RF550
#define MIN_FREQ 100000
#define MAX_FREQ 29999900
#endif

// VFO/MEMORY states.
#define S_VFO 0
#define S_MEM 1

// Button press status.
#define NO_PRESS 0
#define SHORT_PRESS 1
#define MEDIUM_PRESS 2
#define LONG_PRESS 3
#define VERYLONG_PRESS 4

// EEPROM addresses.
#define EEPROM_MAGICADDRESS 0
#define EEPROM_MAGIC 0x55
#define EEPROM_LASTF 1      // Last VFO frequency (4B).
#define EEPROM_LASTM 5      // Last memory (2B).
#define EEPROM_NUMFORMAT 7  // Numeric format (1B).
#define EEPROM_CURSBLINK 8  // Blinking cursor (1B).
#define EEPROM_ZEROSUPP 9   // Non-significant zeros suppression (1B).
#define EEPROM_LCDBLIGHT 10 // LCD backlight ON/OFF (if supported by the display)
// EEPROM addresses 11..95 currently not used
#define EEPROM_MEMS 96      // Memory bank start address (4B*1000=4000B, \
                            // Arduino MEGA EEPROM size is 4096 bytes).

// Serial messages.

// Program versions.
#ifdef TE704C
#define PROGVER "TE704 remote VFO v0.1 (c) 2026 I1EPJ."
#define VERID "v0.1"
#endif
#ifdef RF550
#define PROGVER "RF550 remote VFO v1.1 (c) 2026 I1EPJ."
#define VERID "RF550 Remote VFO v1.1"
#endif

// License.
#define LICENSE "Distributed under GPL v3 or later."

// Command messages.
#define BACKLIGHT "Backlight: "
#define CURSORBLINK "Blinking cursor: "
#define EEEREQUIRED "EEE command required (three uppercase E) to erase EEPROM."
#define INVALIDCMD "Invalid command."
```

```
#define INVALIDFREQ "Invalid frequency."
#define INVALIDMEM "Invalid memory number."
#define INVALIDPARM "Invalid parameter."
#define INVALIDNUM "Numeric parameter required."
#define MEMORYEMPTY "Memory empty."
#define NUMFORMAT "Numeric format: "
#define PARMREQ "Parameter required."
#define ZEROSUPPRS "Non-significant zero suppression: "

// Command results.
#define OK "OK"
#define ERR "ERROR: "

// Help text.
#ifdef BIT_TEST
#define HELP \
"Available serial commands:\n\
B[<n>]        Set/Get blinking cursor status: <n>=0 OFF <n>=1 ON.\n\
C             Copy current memory frequency to VFO and go to VFO mode.\n\
EEE           Init EEPROM (three uppercase E required).\n\
F[<fr>]       Set/Get VFO frequency.\n\
H             Show the help you are reading.\n\
I             Show program info and license.\n\
L[<bkl>]      Set/Get LCD backlight.\n\
M[<nnn>]      Show current/Recall memory <nnn> and go to memory mode.\n\
O             Save options in EEPROM.\n\
R             Check encoder presence and version\n\
S             Show current status.\n\
T             Performs a bit test\n\
U[<n>]        Set/Get numeric format: <n>=0 ITA, <n>=1 USA.\n\
X[<m>]        Change VFO/Memory mode: V=VFO, M=Memory.\n\
W<mnum>[:<f>] Write current VFO frequency/frequency f to memory mnum.\n\
Z[<n>]        Set/Get non-significant zero suppression: <n>=0 OFF <n>=1 ON.\n\n\
Commands can be issued either upper or lower case.\n\
If a command is issued without the optional parameter, current value is shown.\n\
If a command is started and not terminated with CR within 5s, the command is\n\
automatically executed as-is.\n\n"
#else
#define HELP \
"Available serial commands:\n\
B[<n>]        Set/Get blinking cursor status: <n>=0 OFF <n>=1 ON.\n\
C             Copy current memory frequency to VFO and go to VFO mode.\n\
EEE           Init EEPROM (three uppercase E required).\n\
F[<fr>]       Set/Get VFO frequency.\n\
H             Show the help you are reading.\n\
I             Show program info and license.\n\
L[<bkl>]      Set/Get LCD backlight.\n\
M[<nnn>]      Show current/Recall memory <nnn> and go to memory mode.\n\
O             Save options in EEPROM.\n\
R             Check encoder presence and version\n\
S             Show current status.\n\
U[<n>]        Set/Get numeric format: <n>=0 ITA, <n>=1 USA.\n\
X[<m>]        Change VFO/Memory mode: V=VFO, M=Memory.\n\
W<mnum>[:<f>] Write current VFO frequency/frequency f to memory mnum.\n\
Z[<n>]        Set/Get non-significant zero suppression: <n>=0 OFF <n>=1 ON.\n\n\
Commands can be issued either upper or lower case.\n\
If a command is issued without the optional parameter, current value is shown.\n\
If a command is started and not terminated with CR within 5s, the command is\n\
automatically executed as-is.\n\n"
#endif
// LCD messages - 2 lines, 16 char per line max.
//                1234567890123456
#ifdef TE704C
#define LCDCOPY1 "TE704 Remote VFO"
```

```
#endif
#ifdef RF550
#define LCDCOPY1 "RF550 Remote VFO"
#endif
#define LCDCOPY2 "    I1EPJ 2026  "
#define LCDWAIT1 "    Waiting     "
#define LCDWAIT2 " remote enable  "
#define MEMNOTDEF " [NOT DEF]" // 10 characters available (xx,xxx.xxx)

// Check configuration and abort if something wrong.
#if !(defined(RF550) || defined(TE704C))
#error One of RF550 and TE704C must be #defined.
#endif
#if defined(RF550) && defined(TE704C)
#error Only one of RF550 and TE704C must be defined.
#endif

// Function prototypes.
void EEPROM_writeMems(void);
void EEPROM_init(void);
uint64_t uint32_to_bcd(uint32_t usi);
void displayFreq(long f, int x, int y);
void displayMem(int mnum);
int ReadSwitch(void);
void placeCursor(void);
void getCommand(void);
void writeMem(int mnum, long f);
long getMem(int mnum);
void initDisplay(void);
void setFreq(long f);
void checkRemote(void);
#ifdef SAVE_LAST_FREQMEM
void writeLastFM(long f, int m);
long readLastFM(long* fr, int* mn);
#endif

// Global variables.
long f = 14000000, ftmp, fstep, mstep = 1,
     LastFChangeTime, LastFWriteTime,
     LastMChangeTime, LastMWriteTime;
char s[20];
int mnum = 0, mtmp, oldval, newval, delta, state = S_VFO;
#ifdef RF550
//                 -10M-  ----1M----  ---100K---  ---10K----  ----1K----  --100Hz---
//                  2  1  8  4  2  1  8  4  2  1  8  4  2  1  8  4  2  1  8  4  2  1
int bitaddr[22] = {50,52,43,42,44,48,39,38,41,40,35,34,37,36,31,30,33,32,45,49,51,53};
#endif
#ifdef TE704C
// To be checked against interface schematic when available.
//                  -10M- ----1M----  ---100K---  ---10K----  ----1K----  ---100Hz--  ---10Hz---
//                  2  1  8  4  2  1  8  4  2  1  8  4  2  1  8  4  2  1  8  4  2  1  8  4  2  1
int bitaddr[26] =
{22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47};
#endif
bool format_USA = false;          // use USA conventions for digit separators.
bool remote_Enabled;              // flag to signal that RX remote control is enabled.
bool cursor_Blink = false;        // flag to signal a blinking cursor wanted.
bool zero_Suppression = true;     // flag to signal non-significant zero suppression wanted.
bool LCD_Backlight = true;        // LCD Backlight ON/OFF (if supported)

// Peripheral declaratipons.
Encoder enc(ENCODER_CLK, ENCODER_DT);  // Rotary encoder.
LiquidCrystal_PCF8574 lcd(0x27);       // 16x2 LCD display via PCF8574 I2C expander.
```

```
// Define custom characters-
// In the standard HD44780 8 programmable characters are available.

// Short press symbol.
byte shortp[8] = {
  0b00000,
  0b00000,
  0b00000,
  0b00000,
  0b00000,
  0b00000,
  0b00000,
  0b11111
};

// Medium press symbol.
byte mediump[8] = {
  0b00000,
  0b00000,
  0b00000,
  0b00000,
  0b11111,
  0b11111,
  0b11111,
  0b11111
};

// Long press symbol.
byte longp[8] = {
  0b11111,
  0b11111,
  0b11111,
  0b11111,
  0b11111,
  0b11111,
  0b11111,
  0b11111
};

// Waiting hourglass symbol 1 (0°/180°)
byte Wait1[8] = {
  0b00000,
  0b11111,
  0b01110,
  0b00100,
  0b01110,
  0b11111,
  0b00000,
  0b00000
};

// Waiting hourglass symbol 2 (45°/225°)
byte Wait2[8] = {
  0b00000,
  0b00100,
  0b00110,
  0b11111,
  0b01100,
  0b00100,
  0b00000,
  0b00000
};

// Waiting hourglass symbol 3 (90°/270°)
```

```
byte Wait3[8] = {
  0b00000,
  0b10001,
  0b11011,
  0b11111,
  0b11011,
  0b10001,
  0b00000,
  0b00000
};

// Waiting hourglass symbol 4 (135°/315°)
byte Wait4[8] = {
  0b00000,
  0b00100,
  0b01100,
  0b11111,
  0b00110,
  0b00100,
  0b00000,
  0b00000
};

// Copyright symbol (c)
byte Copyright[8] = {
  0b01110,
  0b10001,
  0b10111,
  0b11001,
  0b11001,
  0b10111,
  0b10001,
  0b01110
};

// Initialization.
void setup( ) {

  int i;

  // Pin configuration.
  pinMode(ENCODER_SW, INPUT_PULLUP);
  pinMode(REMOTE, INPUT);

  // Receiver specific configurations.
  #ifdef TE704C
  // Init minimum frequency step (10Hz).
  fstep = 10;
  // Set all 26 BCD lines to output with a LOW value..
  for (i = 0; i < 26; i++) {
    pinMode(bitaddr[i], OUTPUT);
    #ifdef NOT_BUFFERS
    digitalWrite(bitaddr[i], HIGH);
    #else
    digitalWrite(bitaddr[i], LOW);
    #endif
  }
  // TBD if and when hardware supports software LOCAL/REMOTE switch.
  #endif

  #ifdef RF550
  // Init minimum frequency step (100Hz).
  fstep = 100;
  // Set all 22 BCD lines to output.
```

```
for (i = 0; i < 22; i++) {
  pinMode(bitaddr[i], OUTPUT);
  // Set all outputs high to disable open-collector buffers.
  #ifdef NOT_BUFFERS
  digitalWrite(bitaddr[i], LOW);
  #else
  digitalWrite(bitaddr[i], HIGH);
  #endif
}
#endif

#ifdef SAVE_LAST_FREQMEM
// Init timing variables
LastFChangeTime = LastFWriteTime = LastMChangeTime = LastMWriteTime = millis();
#endif

// LCD display setup.
lcd.begin(16,2);
lcd.clear();
lcd.setBacklight(255);

lcd.createChar(1, shortp);
lcd.createChar(2, mediump);
lcd.createChar(3, longp);
lcd.createChar(4, Wait1);
lcd.createChar(5, Wait2);
lcd.createChar(6, Wait3);
lcd.createChar(7, Wait4);
lcd.createChar(8, Copyright);

// Show program name and copyright.
lcd.setCursor(0, 0);
lcd.print(LCDCOPY1);
lcd.setCursor(0, 1);
lcd.print(LCDCOPY2);
lcd.setCursor(2, 1);
lcd.write(char(8));
delay(1000);

// Serial setup.
Serial.begin(115200);
Serial.setTimeout(5000);

// Announce ourselves.
Serial.println(PROGVER);

// Read first encoder value.
oldval = enc.read();

// Init EEPROM if no magic value found.
if (EEPROM.read(EEPROM_MAGICADDRESS) != EEPROM_MAGIC)
  EEPROM_init();

// Read config values.
EEPROM.get(EEPROM_NUMFORMAT, i);
format_USA = (i == 1);

EEPROM.get(EEPROM_CURSBLINK, i);
cursor_Blink = (i == 1);

EEPROM.get(EEPROM_ZEROSUPP, i);
zero_Suppression = (i == 1);

EEPROM.get(EEPROM_LCDBLIGHT, i);
```

```
  LCD_Backlight = (i == 1);

  if (LCD_Backlight)
    lcd.setBacklight(255);
  else
    lcd.setBacklight(0);

  // Wait for the receiver to be in remote mode.
  checkRemote();

  // Setup initial screen.
  initDisplay();
}

// Check if receiver is set to remote mode.
void checkRemote(void) {

  int i, fl;

  // If REMOTE line is low, display a message...
  if (!digitalRead(REMOTE)) {
#ifdef RF550
    // RF550 remote frequency inputs use open collector drivers,
    // so to disable them set all outputs to 1.
    for (i = 0; i < 22; i++) {
      digitalWrite(bitaddr[i], HIGH);
    }
#endif
#ifdef TE704C
// In the TE704C the REMOTE enable switch is in the encoder
// and not in the receiver, so no need to do anything,
// at least until hardware supports software LOCAL/REMOTE
// switching.
#endif

    // Display wait message...
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print(LCDWAIT1);
    lcd.setCursor(0, 1);
    lcd.print(LCDWAIT2);

    // ... and wait until REMOTE line goes high.
    while (!digitalRead(REMOTE)) {
      // While we wait show a rotating hourglass cursor.
      lcd.setCursor(15, 1);
      lcd.write(byte(fl++ % 4 + 4));
      delay(1000);
    }
    // REMOTE gone high, return to normal visualization and exit.
    initDisplay();
  }
}

// Init memory bank in EEPROM (set all memories to -1, memory not defined).
// Called only by EEPROM_init.
void EEPROM_writeMems(void) {
  int addr;

  for (addr = EEPROM_MEMS; addr < EEPROM_MEMS + 4000; addr += 4) {
    EEPROM.put(addr, (long)-1);
  }
}
```

```
// Init EEPROM.
void EEPROM_init(void) {
 // Init magic value
  EEPROM.write(EEPROM_MAGICADDRESS, EEPROM_MAGIC);

  // Init last frequency.
  EEPROM.put(EEPROM_LASTF, (long)MIN_FREQ);

  //Init default options.
  EEPROM.put(EEPROM_NUMFORMAT, false);  // Italian numeric format
  EEPROM.put(EEPROM_CURSBLINK, false);  // No blinking cursor
  EEPROM.put(EEPROM_ZEROSUPP, true);    // Zero suppression on
  // Init all 1000 memories with a [NOT DEF] value.
  EEPROM_writeMems();
  mnum = 0;
}

// Init the display for normal visualization (frequency and memories).
void initDisplay(void) {
  lcd.clear();
  lcd.print("FREQ: ");
  lcd.setCursor(0, 1);
  lcd.print("M000: ");
#ifdef SAVE_LAST_FREQMEM
  readLastFM(&f, &mnum);
#else
  f = MIN_FREQ;
#endif
  displayFreq(f, 6, 0);
  setFreq(f);
  displayMem(mnum);
  lcd.cursor();
  placeCursor();
  if (cursor_Blink)
    lcd.blink();
  else
    lcd.noBlink();
}

// Unsigned long to BCD conversion.
// Bovinely copied from some internet place.
uint64_t uint32_to_bcd(uint32_t usi) {

  uint64_t shift = 16;
  uint64_t result = (usi % 10);

  while (usi = (usi / 10)) {
    result += (usi % 10) * shift;
    shift *= 16;  // weirdly, it's not possible to left shift more than 32 bits.
  }
  return result;
}

// Display the frequency in kilohertz in the first row.
// Format is MM.KKK,H00 e.g. 12.345,600 (italian format) or
// MM,KKK.H00 e.g. 12.345,600 (USA format).
// TE704C minimum step is 10Hz so the hertz digit is always zero.
// RF550 minimum step is 100Hz so the last two digits are always zero.
void displayFreq(long f, int x, int y) {
  int MHz, kHz, Hz;

  lcd.setCursor(x, y);

  MHz = f / 1000000;
```

```
  if (zero_Suppression) {
    if (MHz == 0)
      sprintf(s, "%s", "  ");
    else
      sprintf(s, "%2d", MHz);
  } else {
    sprintf(s, "%02d", MHz);
  }
  lcd.print(s);
  if ((MHz == 0) && (zero_Suppression)) {
    lcd.print(" ");
  } else {
    if (format_USA)
      lcd.print(",");
    else
      lcd.print(".");
  }
  kHz = (f / 1000) % 1000;
  if (zero_Suppression) {
    if (MHz == 0)
      sprintf(s, "%3d", kHz);
    else
      sprintf(s, "%03d", kHz);
  } else {
    sprintf(s, "%03d", kHz);
  }
  lcd.print(s);

  if (format_USA)
    lcd.print(".");
  else
    lcd.print(",");

#ifdef TE704C
  // Make 10 Hz step
  Hz = ((f % 1000) / 10) * 10;
#endif
#ifdef RF550
  // Make 100 Hz step
  Hz = ((f % 1000) / 100) * 100;
#endif
  sprintf(s, "%03d", Hz);
  lcd.print(s);
}

// Display memory number (0..999) and frequency in second row.
void displayMem(int mnum) {

  long f;

  if (zero_Suppression)
    sprintf(s, "%3d", mnum);
  else
    sprintf(s, "%03d", mnum);
  lcd.setCursor(1, 1);
  lcd.print(s);
  f = getMem(mnum);
  if (f > 0)
    displayFreq(f, 6, 1);
  else {
    lcd.setCursor(6, 1);
    lcd.print(MEMNOTDEF);
  }
}
```

```
// Set the frequency in the receiver.
void setFreq(long f) {
  int i, j, b;
  uint64_t fbcd;

  fbcd = uint32_to_bcd(f);
  #ifdef NOT_BUFFERS
  // Inverting buffers used, so invert bits.
  fbcd = ~fbcd;
  #endif
  // fbcd is MMKKKHHH where a digit is 4 bit BCD value,
  // so total length = 30 bits, since the tens of MHz digit is only 2 bits long.
  #ifdef TE704C
  // Frequency LSB is the 10 Hz digit.
  // So we we discard the least significant 4 bits and have 26 BCD bits to write.
  for (i = 4, j = 25; i < 30; i++, j--) {
  #endif
  #ifdef RF550
  // Frequency LSB is the 100 Hz digit.
  // So we we discard the least significant 8 bits and have 22 BCD bits to write.
  for (i = 8, j = 21; i < 30; i++, j--) {
  #endif
    // Get bit value.
    b = (fbcd >> i) & 1;
    // Finally write the bit to the appropriate output line.
    digitalWrite(bitaddr[j], b);
  }
}

// Read encoder switch.
int ReadSwitch(void) {

  long st, et, dt;
  int ypos;

  ypos = (state == S_VFO) ? 0 : 1;
  if (digitalRead(ENCODER_SW) == 0) {
    delay(50);                              // wait bounce time
    if (digitalRead(ENCODER_SW) == 0) {  // switch still closed, valid press
      st = millis();
      lcd.noCursor();
      lcd.setCursor(5, ypos);
      lcd.write(byte(1));  // short press symbol
      while (digitalRead(ENCODER_SW) == 0) {
        if ((millis() - st) > LPRESS_TIME) {
          lcd.setCursor(5, ypos);
          lcd.print("B");  // Very long press = backlight
        }
        if ((millis() - st) > MPRESS_TIME) {
          lcd.setCursor(5, ypos);
          lcd.write(byte(3));  // long press symbol
        } else if ((millis() - st) > SPRESS_TIME) {
          lcd.setCursor(5, ypos);
          lcd.write(byte(2));  // medium press symbol
        }
      }
      lcd.setCursor(5, ypos);
      lcd.print(" ");
      placeCursor();
      et = millis();
      dt = et - st;
      if (dt > LPRESS_TIME)
        return VERYLONG_PRESS;
```

```
      if (dt > MPRESS_TIME)
        return LONG_PRESS;
      if (dt > SPRESS_TIME)
        return MEDIUM_PRESS;
      if (dt < SPRESS_TIME)
        return SHORT_PRESS;
    }
  }
  lcd.cursor();
  return NO_PRESS;
}

// Write f to memory mnum.
void writeMem(int mnum, long f) {
  int addr;

  addr = EEPROM_MEMS + 4 * mnum;
  EEPROM.put(addr, f);
}

// Get frequency from memory mnum.
long getMem(int mnum) {
  int addr;
  long val;

  addr = 4 * mnum + EEPROM_MEMS;
  EEPROM.get(addr, val);
  return val;
}

// Place the LCD cursor under the frequency or memory digit to be modified.
void placeCursor(void) {
  long fstmp;
  int xpos, ypos;

  if (state == S_VFO) {
    // Start at hertz units position
    xpos = 15;

    // For every decimal digit in fstep move left a column.
    fstmp = fstep;
    while (fstmp > 1) {
      fstmp /= 10;
      xpos--;
    }

    // Adjust xpos to account for separators.
    if (xpos < 13) xpos--;
    if (xpos < 9) xpos--;
    // VFO line is in the first row.
    ypos = 0;
  }
  if (state == S_MEM) {
    switch (mstep) {
      case 1: {
        xpos = 3;
        break;
      }
      case 10: {
        xpos = 2;
        break;
      }
      case 100: {
        xpos = 1;
```

```
        break;
      }
    }
    // Memory line is in the second row.
    ypos = 1;
  }
  // Place and show cursor.
  lcd.setCursor(xpos, ypos);
  lcd.cursor();
}

#ifdef SAVE_LAST_FREQMEM
// Write once last frequency and/or memory number to EEPROM after
// 1' of inactivity.
void writeLastFM(long f, int m) {
  unsigned long now = millis();
  if ((now - LastFChangeTime) > 60000 && LastFWriteTime != LastFChangeTime) {
    LastFWriteTime = LastFChangeTime;
    EEPROM.put(EEPROM_LASTF, f);
  }
  if ((now - LastMChangeTime) > 60000 && LastMWriteTime != LastMChangeTime) {
    LastMWriteTime = LastMChangeTime;
    EEPROM.put(EEPROM_LASTM, m);
  }
}

// Read last frequency and memory from EEPROM.
long readLastFM(long* fr, int* mn) {
  EEPROM.get(EEPROM_LASTF, *fr);
  EEPROM.get(EEPROM_LASTM, *mn);
}
#endif

// Get a command from serial (USB) port.
// Serial format is 115200 baud, 8 bit, no parity.
// Commands can be upper or lower case, except the EEE command.
void getCommand(void) {
  char cmd[41], buf[80];
  const char* errDescr;
  int nc, xpos;
  bool resultOK;

  if (Serial.available()) {
    nc = Serial.readBytesUntil(0x0d, cmd, 40);
    cmd[nc] = 0;
    resultOK = true;
    switch (toupper(cmd[0])) {
      case '\0': {
        break;
      }
      case 'B': {
        // Blinking cursor ON/OFF.
        // Syntax:
        // B[<blinking cursor state, 0=OFF 1=ON>]
        // Without parameter shows current value.
        if (cmd[1] != 0) {
          cursor_Blink = (cmd[1] == '1');
        }
        Serial.print(CURSORBLINK);
        Serial.println(cursor_Blink ? "ON" : "OFF");
        if (cursor_Blink)
          lcd.blink();
        else
          lcd.noBlink();
```

```
    break;
  }
  case 'C': {
    // Copy current memory frequency to VFO and go to VFO mode.
    // No parameters required.
    // Syntax:
    // C
    state = S_VFO;
    f = getMem(mnum);
    setFreq(f);
    displayFreq(f, 6, 0);
    placeCursor();
    sprintf(buf, "S0,F%8.8ld", f);
    Serial.println(buf);
    break;
  }
  case 'E': {
    // Init EEPROM.
    // !!! THIS WILL ERASE ALL MEMORIES !!!
    // To be sure writing E exactly three times is required.
    // No parameters required.
    // Syntax:
    // EEE
    if (cmd[0] == 'E' && cmd[1] == 'E' && cmd[2] == 'E' && cmd[3] == '\0') {
      EEPROM_init();
      if ((ftmp = getMem(mnum)) > 0)
        sprintf(buf, "S%s,M%3.3d:%8.8ld", (state==S_VFO)?"0":"1", mnum, ftmp);
      else
        sprintf(buf, "S%s,M%3.3d: [NOT DEF]", (state==S_VFO)?"0":"1", mnum);
      Serial.println(buf);
    } else {
      errDescr = EEEREQUIRED;
      resultOK = false;
    }
    break;
  }
  case 'F': {
    // Set/Get VFO frequency.
    // Syntax:
    // F[<frequency in Hertz>]
    // Without parameter shows current value.
    if (cmd[1] != 0) {
      for (char* p = &cmd[1]; *p != 0; p++)
        resultOK &= isdigit(*p);
      if (resultOK) {
        #ifdef TE704C
        ftmp = 10 * (atol(&cmd[1]) / 10);
        #endif
        #ifdef RF550
        ftmp = 100 * (atol(&cmd[1]) / 100);
        #endif
        if (ftmp >= MIN_FREQ && ftmp <= MAX_FREQ) {
          f = ftmp;
          state = S_VFO;
          displayFreq(f, 6, 0);
          setFreq(f);
          placeCursor();
          LastFChangeTime = millis();
        } else {
          errDescr = INVALIDFREQ;
          resultOK = false;
        }
      } else {
        errDescr = INVALIDNUM;
```

```
      resultOK = false;
    }
  }
  if (resultOK) {
    sprintf(buf, "F%8.8ld", f);
    Serial.println(buf);
  }
  break;
}
case 'H': {
  // Display help (short command list).
  // No parameters required.
  // Syntax: H
  Serial.print(HELP);
  resultOK = true;
  break;
}
case 'I': {
  // Shows program info and license.
  // No parameters required.
  // Syntax:
  // I
  Serial.println(PROGVER);
  Serial.println(LICENSE);
  break;
}
case 'L': {
  // LCD backlight on/off
  // 0=OFF, 1=ON
  // Syntax:
  // L[<LCD backlight>]
  // Without parameter shows current state
  if (cmd[1] != 0) {
    if (cmd[1] == '0' || cmd[1] == '1') {
      LCD_Backlight = (cmd[1] == '1');
      if (LCD_Backlight)
        lcd.setBacklight(255);
      else
        lcd.setBacklight(0);
    } else {
      errDescr = INVALIDPARM;
      resultOK = false;
    }
  }
  Serial.print(BACKLIGHT);
  Serial.println((LCD_Backlight)?"ON":"OFF");
  break;
}
case 'M': {
  // If memory not empty, recalls it and goes to memory mode.
  // Syntax:
  // M[<memory number, 0...999>[-]]
  // If - is appended after memory number, reads memory content
  // without sending it to the receiver.
  // Without parameter shows current value.
  char* p;
  if (cmd[1] != 0) {
    for (p = &cmd[1]; *p != 0; p++)
      resultOK &= (isdigit(*p) || (*p='-'));
    if (resultOK) {
      mtmp = atoi(&cmd[1]);
      if (mtmp >= 0 && mtmp < 1000) {
        ftmp = getMem(mtmp);
          if (strchr(cmd,'-') == NULL) {
```

```
            state = S_MEM;
            displayMem(mtmp);
            if (ftmp > 0) setFreq(ftmp);
            placeCursor();
            LastMChangeTime = millis();
            mnum = mtmp;
          }
        } else {
          errDescr = INVALIDMEM;
          resultOK = false;
        }
      } else {
        resultOK = false;
        errDescr = INVALIDNUM;
      }
    }
    if ((ftmp = getMem(mtmp)) > 0)
      sprintf(buf, "S%s,M%3.3d:%8.8ld", (state==S_VFO)?"0":"1", mtmp, ftmp);
    else
      sprintf(buf, "S%s,M%3.3d: [NOT DEF]", (state==S_VFO)?"0":"1", mtmp);
    Serial.println(buf);
    break;
  }
  case 'O': {
    // Save options in EEPROM.
    // No parameters required.
    // Syntax:
    // O
    EEPROM.put(EEPROM_NUMFORMAT, (format_USA ? 1 : 0));
    EEPROM.put(EEPROM_CURSBLINK, (cursor_Blink ? 1 : 0));
    EEPROM.put(EEPROM_ZEROSUPP, (zero_Suppression ? 1 : 0));
    EEPROM.put(EEPROM_LCDBLIGHT, (LCD_Backlight ? 1 : 0));
    break;
  }
  case 'R': {
    // Do nothing command, used by RF550Control to check
    // for encoder ready. Returns RRR:<version>.
    Serial.println("RRR:" VERID);
    break;
  }
  case 'S': {
    // Return current status as
    // S<m>,F<freq>,M<mem#>:<memf>,U<USAfmt>,Z<zerosupp>,B<curs blink>,L<LCD bklght>
    // Exanple:
    // No parameters required.
    // Syntax:
    // S
    if ((ftmp = getMem(mnum)) > 0)
      sprintf(buf,"S%c,F%8.8ld,M%3.3d:%8.8ld,U%d,Z%d,B%d,L%d",
      (state==S_VFO)?'0':'1',f,mnum,ftmp,
      (format_USA)?1:0,(zero_Suppression)?1:0,(cursor_Blink)?1:0,(LCD_Backlight)?1:0);
    else
      sprintf(buf,"S%c,F%8.8ld,M%3.3d:[NOT DEF],U%d,Z%d,B%d,L%d",
      (state==S_VFO)?'0':'1',f,mnum,
      (format_USA)?1:0,(zero_Suppression)?1:0,(cursor_Blink)?1:0,(LCD_Backlight)?1:0);
    Serial.println(buf);
    break;
  }
  #ifdef BIT_TEST
  case 'T': {
    // For test purposes only
    // Sets to 1 then to 0 in turn all bits.
    // No parameters required.
    // Syntax:
```

```
    // T
    int i;
    #ifdef RF550
    for (i=0; i<22; i++)
      // clear all bits
      digitalWrite(bitaddr[i], LOW);
    for (i=0; i<22; i++) {
      // Set bit
      digitalWrite(bitaddr[i], HIGH);
      delay(500);
      // Reset bit
      digitalWrite(bitaddr[i], LOW);
    }
    #endif
    #ifdef TE704C
    for (i=0; i<26; i++)
      // clear all bits
      digitalWrite(bitaddr[i], LOW);
    for (i=0; i<26; i++) {
      // Set bit
      digitalWrite(bitaddr[i], HIGH);
      delay(500);
      // Reset bit
      digitalWrite(bitaddr[i], LOW);
    }
    #endif
  }
  #endif
  case 'U': {
    // Sets/returns numeric format.
    // Without parameter shows current value
    // Syntax: U[<Numeric format>]
    // <numeric format>=1 USA format (MM,KKK.HHH)
    // <numeric format>=0 ITA format (MM.KKK,HHH)
    if (cmd[1] != 0) {
      if (cmd[1] == '0' || cmd[1] == '1') {
        format_USA = (cmd[1] == '1');
        // Update display
        displayFreq(f, 6, 0);
        displayMem(mnum);
      } else {
        errDescr = INVALIDPARM;
        resultOK = false;
      }
    }
    if (resultOK) {
      Serial.print(NUMFORMAT);
      Serial.println(format_USA ? "USA" : "ITA");
    }
    break;
  }
  case 'W': {
    // Write current VFO frequency/new frequency to memory mnum
    // Syntax: W<mnum>[:<new frequency>], mnum is 0...999, <new frequency> is the frequency
    in Hz
    if (cmd[1] != 0) {
      mtmp = atoi(&cmd[1]);
      if (mtmp >= 0 && mtmp < 1000 && cmd[1] != 0) {
        char* p = strstr(cmd,":");
        if (p==NULL) {
          mnum = mtmp;
          writeMem(mnum, f);
        } else {
          ftmp = atol(p+1);
```

```
        writeMem(mtmp,ftmp);
      }
      delay(100);
      sprintf(buf, "S1,M%3.3d:%8.8ld", mnum, getMem(mnum));
      Serial.println(buf);
      displayMem(mnum);
      placeCursor();
    } else {
      errDescr = INVALIDMEM;
      resultOK = false;
    }
  } else {
    errDescr = PARMREQ;
    resultOK = false;
  }
  break;
}
case 'X': {
  // Get/Set VFO or Memory mode.
  // Syntax: X[<mode>]
  // <mode> can be V for VFO and M for memory.
  if (cmd[1] != 0) {
    char c = toupper(cmd[1]);
    if (c == 'V' || c == 'M') {
      if (c == 'V') {
        state = S_VFO;
        setFreq(f);
      }
      if (c == 'M') {
        state = S_MEM;
        setFreq(getMem(mnum));
      }
      // Update display
      displayFreq(f, 6, 0);
      displayMem(mnum);
      placeCursor();
    } else {
      errDescr = INVALIDPARM;
      resultOK = false;
    }
  }
  if (resultOK) {
    sprintf(buf, "S%c", (state == S_VFO) ? 'V' : 'M');
    Serial.println(buf);
  }
  break;
}
case 'Z': {
  // Sets/shows non-significant zeros suppression option.
  // Syntax: Z[<zero suppression>]
  // <zero suppression>=1 non-significant zero suppression ON
  // <zero suppression>=0 non-significant zero suppression OFF
  // Without parameter shows current value.
  if (cmd[1] != 0) {
    if (cmd[1] == '0' || cmd[1] == '1') {
      zero_Suppression = (cmd[1] == '1');
      // Update display
      displayFreq(f, 6, 0);
      displayMem(mnum);
      resultOK = true;
    }
  }
  if (resultOK) {
    Serial.print(ZEROSUPPRS);
```

```
          Serial.println(zero_Suppression ? "ON" : "OFF");
        }
        break;
      }
      default: {
        // Unrecognized command.
        resultOK = false;
        errDescr = INVALIDCMD;
      }
    }
    // If last command was succesful, print OK.
    if (resultOK) {
      Serial.println("OK");
    } else {
      // Issue and error message.
      Serial.print(ERR);
      Serial.println(errDescr);
    }
  }
}

// Forever loop.
void loop( ) {

  int xpos, sw;
  long ftmp;

  // Check if the REMOTE line is still high.
  // If not, save last frequency and memory in EEPROM and call the
  // checkRemote( ) function which disables frequency outputs and blocks
  // until the REMOTE line goes high again.
  if (!digitalRead(REMOTE)) {
    EEPROM.put(EEPROM_LASTF, f);
    EEPROM.put(EEPROM_LASTM, mnum);
  }
  // Block calling the checkRemote( ) function.
  checkRemote( );

  // Read & execute remote command.
  getCommand( );

  // Read encoder.
  newval = enc.read( );
  delta = newval - oldval;
  oldval = newval;

  // Read encoder switch.
  sw = ReadSwitch( );

  // VFO/Memory mode state machine.
  switch (state) {
    case S_VFO: {
      switch (sw) {
        case SHORT_PRESS: {
          // change frequency step
          fstep *= 10;
          #ifdef TE704C
          if (fstep > 10000000) fstep = 10;
          #endif
          #ifdef RF550
          if (fstep > 10000000) fstep = 100;
          #endif
          displayFreq(f, 6, 0);
          placeCursor( );
```

```
      break;
    }
    case MEDIUM_PRESS:  {
      // switch to memory mode
      state = S_MEM;
      displayMem(mnum);
      setFreq(getMem(mnum));
      placeCursor();
      break;
    }
    case LONG_PRESS: {
      // copy memory frequency to VFO
      ftmp = getMem(mnum);
      if (ftmp > 0) {
        f = ftmp;
        setFreq(f);
        displayFreq(f, 6, 0);
        placeCursor();
      }
      break;
    }
    case VERYLONG_PRESS: {
      LCD_Backlight = !LCD_Backlight;
      if (LCD_Backlight)
        lcd.setBacklight(255);
      else
        lcd.setBacklight(0);
    }
  }
  if (delta > 1) {
    f += fstep;
    if (f > MAX_FREQ) f = MAX_FREQ;
  }
  if (delta < -1) {
    f -= fstep;
    if (f < MIN_FREQ) f = MIN_FREQ;
  }
  if (delta != 0) {
    // Update display.
    displayFreq(f, 6, 0);
    setFreq(f);
    placeCursor();
    #ifdef SAVE_LAST_FREQMEM
    LastFChangeTime = millis();
    #endif
  }
  break;
}
case S_MEM: {
  switch (sw) {
    case SHORT_PRESS: {
      // Changes memory number digit to be modified
      mstep *= 10;
      if (mstep > 100) mstep = 1;
      break;
    }
    case MEDIUM_PRESS: {
      // Switches to VFO mode
      state = S_VFO;
      setFreq(f);
      placeCursor();
      break;
    }
    case LONG_PRESS: {
```

```
          // Writes VFO frequency to current memory
          writeMem(mnum, f);
          displayMem(mnum);
          setFreq(getMem(mnum));
          break;
        }
        case VERYLONG_PRESS: {
          LCD_Backlight = !LCD_Backlight;
          if (LCD_Backlight)
            lcd.setBacklight(255);
          else
            lcd.setBacklight(0);
        }
      }
      if (delta > 1) mnum += mstep;
      if (delta < -1) mnum -= mstep;
      if (mnum > 999) mnum = 0;
      if (mnum < 0) mnum = 999;
      if (delta != 0) {
        // Update display.
        displayMem(mnum);
        if ((ftmp = getMem(mnum)) > 0) {
          setFreq(ftmp);
          LastMChangeTime = millis();
        }
      }
      // place cursor under the memory digit to be modified.
      placeCursor();
      break;
    }
  }
  #ifdef SAVE_LAST_FREQMEM
  writeLastFM(f, mnum);
  #endif
  delay(50);
}
```