

monitor_blackout.ino

```
/*
=====
MONITOR BLACKOUT DI RETE v2 - Arduino Nano + SIM800L + PC817 + DS3231
=====
officinasottocasa.it

Novità rispetto alla v1:
- Lettura rete via PC817 in configurazione open-collector con
  pull-up INTERNA di Arduino (INPUT_PULLUP), senza partitore esterno.
- Filtro RC hardware (R3 100k + C3 100nF) tra uscita PC817 e D3.
- RTC DS3231 (I2C) per timestamp reali e debounce indipendente da millis().
- Sleep mode (SLEEP_MODE_PWR_DOWN) + Pin Change Interrupt su D3 + Watchdog (~8s).
  (INT1 in modalita' CHANGE NON sveglia l'ATmega328P da PWR_DOWN: in quello
  stato funziona solo l'interrupt a livello LOW. Il Pin Change si'.)
- SMS in arrivo letti a polling (AT+CMGL) ad ogni risveglio, non più
  intercettati "al volo" (incompatibile con lo sleep).
- Controllo periodico della tensione di batteria, con SMS di avviso.
- Collegamento SIM800L TX/RX ad Arduino tramite modulo level shifter
  logico bidirezionale 3.3V/5V (non più con partitore resistivo).
- Display OLED 128x64 (controller SSD1306, I2C) con data e ora, stato della
  rete (ONLINE/OFFLINE), tensione di batteria, stato di Arduino (ATTIVO/
  SLEEP), del SIM800L (ATTIVO/ASSENTE) e della sua connessione alla rete GSM
  (CONNESSO/RICERCA/NO RETE). Libreria U8g2 in modalita' U8x8 (solo
  testo, nessun buffer in RAM: il buffer da 1 KB di Adafruit_SSD1306 non ci
  sta in un ATmega328P insieme al resto dello sketch).
- L'RTC viene aggiornato all'ora di compilazione ad ogni nuovo upload dello
  sketch (riconosciuto con un marcatore in EEPROM), non ad ogni reset.
- Lettura della batteria indipendente dalla tensione di alimentazione di
  Arduino: a ogni lettura la Vcc reale viene misurata con il riferimento
  interno da 1.1V (bandgap), invece di supporre che sia esattamente 5.0V.

Richiede le librerie "RTCLib" di Adafruit e "U8g2" di olikraus
(Library Manager IDE Arduino).

COLLEGAMENTO DISPLAY OLED (I2C, sullo stesso bus del DS3231):
  OLED GND -> GND
  OLED VCC -> 5V (o 3V3, secondo il modulo)
  OLED SDA -> A4 (in comune con SDA del DS3231)
  OLED SCL -> A5 (in comune con SCL del DS3231)
Indirizzo I2C tipico 0x3C (0x3D se il modulo ha il ponticello spostato).
Il DS3231 usa 0x68: nessun conflitto. Se il display non risponde, lo
sketch funziona comunque (il monitor blackout non dipende dallo schermo).

NOTE IMPORTANTI (invariate dalla v1):
- Il SIM800L va alimentato DIRETTAMENTE dalla batteria/step-down
  (3.7-4.2V), MAI dal pin 5V di Arduino: assorbe picchi fino a 2A.
- Condensatore C1 (2200uF) il piu' vicino possibile ai pin di
  alimentazione del SIM800L, con C2 (100nF) in parallelo.
- Le linee seriali TX/RX del SIM800L lavorano a 3.3V: il livello
  è tradotto dal modulo level shifter (lato HV=5V, lato LV=3V3).
=====
*/

#include <SoftwareSerial.h>
#include <Wire.h>
#include <EEPROM.h>
#include <RTCLib.h>
#include <U8x8lib.h>
#include <avr/sleep.h>
#include <avr/wdt.h>

// ----- CONFIGURAZIONE -----

const uint8_t PIN_RETE = 3; // PCINT19 (gruppo PCINT2) su Arduino Nano/Uno
const uint8_t SIM800_RX = 7; // <- TX del SIM800L (via level shifter)
const uint8_t SIM800_TX = 8; // -> RX del SIM800L (via level shifter)
const uint8_t PIN_BATTERIA = A0;

// Numeri di telefono a cui inviare gli SMS di allarme (formato internazionale)
const char* NUMERO_ALLARME_1 = "+393403638861";
```

```

const char* NUMERO_ALLARME_2 = "+393394071918";

// Invia un SMS di conferma all'accensione del dispositivo
const bool SMS_ALL_AVVIO = true;

// Debounce: la rete deve rimanere nel nuovo stato per almeno questo
// tempo (secondi) prima di essere considerata un evento confermato.
const uint16_t DEBOUNCE_SEC = 8;

// Ogni quanti risvegli da Watchdog (~8s l'uno) controllare la batteria.
// 40 risvegli x 8s = ~5 minuti.
const uint16_t BATTERIA_OGNI_N_WAKE = 40;

const float R_ALTO = 10000.0;
const float R_BASSO = 10000.0;
// Riferimento interno (bandgap) dell'ATmega328P usato per misurare la batteria.
// Il valore nominale e' 1.1V ma da chip a chip puo' scostarsi fino a circa
// il 10%: se lo sketch e il tester non concordano, si regola questo valore
// (VREF_INTERNO = 1.1 * V_tester / V_mostrata). La tensione di Arduino
// (Vcc) invece non serve piu': viene misurata a ogni lettura.
// Taratura attuale: con 1.1 lo sketch leggeva 4.26V contro 4.16V del tester,
// quindi 1.1 * 4.16 / 4.26 = 1.074.
const float VREF_INTERNO = 1.074;
const uint8_t LETTURE_ADC = 8; // letture mediate per ogni misura, riduce il rumore
const float SOGLIA_BATTERIA_BASSA = 3.5; // volt
const float ISTERESI_BATTERIA = 0.15; // volt, evita SMS ripetuti

// Display OLED SSD1306 128x64 (I2C)
const uint8_t OLED_INDIRIZZO = 0x3C; // 0x3D se il modulo ha il ponticello spostato
const uint8_t OLED_CONTRASTO = 64; // 0-255: piu' basso = meno consumo e meno usura

// Aggiornamento RTC ad ogni upload: secondi che passano tra la compilazione
// e l'avvio dello sketch (upload + reset), sommati all'ora di compilazione.
// E' una stima: l'errore residuo e' dell'ordine di qualche decina di secondi.
const uint8_t RTC_RITARDO_UPLOAD_SEC = 10;
const int EEPROM_INDIRIZZO_COMPILAZIONE = 0; // occupa 4 byte (uint32_t)

// Valore di statoGSM quando il SIM800L non risponde (gli altri valori sono
// quelli restituiti da AT+CREG?: 0 non registrato, 1 registrato, 2 in
// ricerca, 3 negato, 5 registrato in roaming)
const uint8_t STATO_GSM_NESSUNA_RISPOSTA = 255;

// Esito di una risposta del SIM800L (vedi esitoRisposta)
const uint8_t RISPOSTA_INCOMPLETA = 0;
const uint8_t RISPOSTA_OK = 1;
const uint8_t RISPOSTA_ERRORE = 2;

// ----- OGGETTI GLOBALI -----

SoftwareSerial sim800(SIM800_RX, SIM800_TX);
RTC_DS3231 rtc;
U8X8_SSD1306_128X64_NONAME_HW_I2C oled(U8X8_PIN_NONE);

// ----- VARIABILI DI STATO -----

volatile bool flagWdtInterrupt = false;

bool statoReteGrezzo = true; // ultima lettura istantanea del pin
bool statoReteConfermato = true; // true = rete presente (assunzione iniziale)
bool cambioInCorso = false;

DateTime timestampPrimoCambio;
DateTime timestampInizioBlackout;

uint16_t contatoreWake = 0;
bool batteriaBassaSegnalata = false;
bool oledPresente = false;
bool sim800Attivo = false; // true = il SIM800L risponde ai comandi AT
uint8_t statoGSM = STATO_GSM_NESSUNA_RISPOSTA; // registrazione alla rete GSM (AT+CREG?)

// ----- SETUP -----

void setup() {

```

```

Serial.begin(9600);
sim800.begin(9600);
Wire.begin();

pinMode(PIN_RETE, INPUT_PULLUP);

inizializzaOLED();

if (!rtc.begin()) {
    Serial.println(F("Errore: RTC DS3231 non rilevato."));
} else {
    sincronizzaRtcAlNuovoUpload();
}

inizializzaSIM800();
aggiornaStatoGSM();

statoReteGrezzo = (digitalRead(PIN_RETE) == LOW);
statoReteConfermato = statoReteGrezzo;

// Pin Change Interrupt su D3: sveglia il micro da PWR_DOWN ad ogni cambio di
// stato (sia blackout che ritorno rete). NON si definisce l'ISR: il vettore
// PCINT2_vect e' gia' definito da SoftwareSerial (un secondo ISR darebbe
// errore di linker per definizione multipla) e basta che l'interrupt scatti
// per riportare il micro fuori dallo sleep. Il lavoro vero lo fa loop().
PCIFR |= _BV(digitalPinToPCICRbit(PIN_RETE));
*digitalPinToPCMSK(PIN_RETE) |= _BV(digitalPinToPCMSKbit(PIN_RETE));
*digitalPinToPCICR(PIN_RETE) |= _BV(digitalPinToPCICRbit(PIN_RETE));
impostaWatchdog();

aggiornaDisplay();

if (SMS_ALL_AVVIO) {
    DateTime ora = rtc.now();
    String msg = "Sistema monitor blackout avviato il ";
    msg += formattaData(ora) + ". Rete: ";
    msg += statoReteConfermato ? "presente." : "ASSENTE.";
    inviaSMSATutti(msg);
}
}

// ----- WATCHDOG -----

ISR(WDT_vect) {
    flagWdtInterrupt = true;
}

void impostaWatchdog() {
    cli();
    wdt_reset();
    MCUSR &= ~(1 << WDRF);
    WDTCR |= (1 << WDCE) | (1 << WDE);
    // ~8 secondi, modalita' solo interrupt (nessun reset del micro)
    WDTCR = (1 << WDIE) | (1 << WDP3) | (1 << WDP0);
    sei();
}

// ----- SLEEP -----

void vaiInSleep() {
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);
    sleep_enable();
    sleep_cpu(); // il micro si ferma qui fino al prossimo interrupt
    sleep_disable();
}

// ----- LOOP -----

void loop() {
    // Il display OLED resta acceso anche durante lo sleep: lo stato di Arduino
    // va scritto PRIMA di addormentarsi e riscritto appena svegli.
    oledStatoArduino(false);

```

```

// Si esce dallo sleep per: cambio di stato su D3 (Pin Change), Watchdog
// (~8s) o attivita' sulla linea RX del SIM800L.
vaiInSleep();

oledStatoArduino(true);

bool risvegliowdt = flagWdtInterrupt;
flagWdtInterrupt = false;
if (risvegliowdt) contatoreWake++;

gestisciRete();
controllaSMSNonLetti();
aggiornaStatoGSM();

if (risvegliowdt && contatoreWake % BATTERIA_OGNI_N_WAKE == 0) {
    controllaBatteria();
}

aggiornaDisplay();
}

// ----- RTC: AGGIORNAMENTO AD OGNI UPLOAD -----

// Porta l'RTC all'ora di compilazione quando lo sketch e' stato appena
// caricato. Il riconoscimento e' fatto con un marcatore (l'ora di
// compilazione) salvato in EEPROM: se coincide, questo e' un semplice
// riavvio e l'RTC NON va toccato. Altrimenti, dopo ogni riavvio (per esempio
// dopo un blackout) l'ora tornerebbe indietro a quella dell'ultimo upload.
// Se l'IDE non ricompila lo sketch, __TIME__ resta quello della compilazione
// precedente: per forzare l'aggiornamento basta modificare lo sketch.
void sincronizzaRtcAlNuovoUpload() {
    DateTime compilazione(F(__DATE__), F(__TIME__));
    uint32_t marcatoreCompilazione = compilazione.unixtime();

    uint32_t marcatoreSalvato;
    EEPROM.get(EEPROM_INDIRIZZO_COMPILAZIONE, marcatoreSalvato);

    // rtc.lostPower(): batteria tampone del modulo scarica/assente
    if (marcatoreSalvato != marcatoreCompilazione || rtc.lostPower()) {
        rtc.adjust(compilazione + TimeSpan(RTC_RITARDO_UPLOAD_SEC));
        EEPROM.put(EEPROM_INDIRIZZO_COMPILAZIONE, marcatoreCompilazione);
    }
}

// ----- LOGICA RETE / DEBOUNCE (basato su RTC) -----

void gestisciRete() {
    bool letturaAttuale = (digitalRead(PIN_RETE) == LOW); // true = rete presente
    DateTime ora = rtc.now();

    if (letturaAttuale == statoReteConfermato) {
        // stato coerente con quello confermato: nessun cambiamento in corso
        cambioInCorso = false;
        statoReteGrezzo = letturaAttuale;
        return;
    }

    if (!cambioInCorso || letturaAttuale != statoReteGrezzo) {
        // primo rilevamento del cambiamento (o "rimbalzo" nel frattempo)
        cambioInCorso = true;
        statoReteGrezzo = letturaAttuale;
        timestampPrimoCambio = ora;
        return;
    }

    // il nuovo stato persiste: verifichiamo da quanto tempo
    TimeSpan trascorso = ora - timestampPrimoCambio;
    if (trascorso.totalseconds() >= DEBOUNCE_SEC) {
        statoReteConfermato = letturaAttuale;
        cambioInCorso = false;

        // Il display si aggiorna subito, prima dell'invio degli SMS (che dura
        // diversi secondi).
    }
}

```

```

    aggiornaDisplay();

    if (!statoReteConfermato) {
        // INIZIO BLACKOUT
        timestampInizioBlackout = ora;
        String msg = "ATTENZIONE: blackout rilevato alle ";
        msg += formattaData(ora);
        inviaSMSATutti(msg);
    } else {
        // RIPRISTINO
        TimeSpan durata = ora - timestampInizioBlackout;
        long minuti = durata.totalSeconds() / 60;
        String msg = "Rete ripristinata alle " + formattaData(ora);
        msg += ". Durata blackout: circa " + String(minuti) + " minuti.";
        inviaSMSATutti(msg);
    }
}

// Formatta una DateTime dell'RTC in un testo leggibile per gli SMS
String formattaData(const DateTime& d) {
    char buf[20];
    sprintf(buf, sizeof(buf), "%02d/%02d/%04d %02d:%02d",
        d.day(), d.month(), d.year(), d.hour(), d.minute());
    return String(buf);
}

// ----- DISPLAY OLED -----

// Scrive una riga di testo (caratteri 8x8 pixel, 16 per riga, 8 righe)
// riempiendola di spazi: cosi' non serve cancellare lo schermo tra un
// aggiornamento e l'altro e non c'e' sfarfallio.
// Versione per testo in RAM (buffer costruiti a runtime)...
void oledRiga(uint8_t riga, const char* testo) {
    char buf[17];
    sprintf(buf, sizeof(buf), "%-16s", testo);
    oled.drawString(0, riga, buf);
}

// ...e per testo fisso F("..."), che resta in flash e non occupa RAM
void oledRiga(uint8_t riga, const __FlashStringHelper* testo) {
    char buf[17];
    strncpy_P(buf, (PGM_P)testo, sizeof(buf) - 1);
    buf[sizeof(buf) - 1] = '\0';
    oledRiga(riga, buf);
}

void inizializzaOLED() {
    // begin() non verifica che il display sia collegato: lo cerchiamo noi sul bus.
    Wire.beginTransmission(OLED_INDIRIZZO);
    oledPresente = (Wire.endTransmission() == 0);
    if (!oledPresente) return;

    oled.setI2CAddress(OLED_INDIRIZZO << 1); // U8x8 vuole l'indirizzo a 8 bit
    oled.begin();
    oled.setContrast(OLED_CONTRASTO);
    oled.setFont(u8x8_font_chroma48medium8_r);
    oledRiga(3, F("Avvio in corso"));
}

// Stato di Arduino (riga 5). Quando il display e' aggiornato Arduino e'
// sveglio per definizione: "SLEEP" e' quello che si legge durante lo sleep,
// perche' l'SSD1306 continua a mostrare l'ultima immagine da solo.
void oledStatoArduino(bool attivo) {
    if (!oledPresente) return;
    oledRiga(5, attivo ? F("Arduino: ATTIVO") : F("Arduino: SLEEP"));
}

// Disegna l'intero schermo (8 righe da 8 pixel):
// 0    data e ora (hh:mm: lo schermo si aggiorna ad ogni risveglio, circa
//      ogni 8 secondi, quindi i secondi non avrebbero senso)
// 1-3  stato della rete monitorata dal PC817 (ONLINE/OFFLINE, doppia dimensione)
// 4    tensione di batteria

```

```

// 5 stato di Arduino
// 6 SIM800L: ATTIVO (risponde ai comandi AT) o ASSENTE (non risponde:
// non collegato, spento o velocita' seriale diversa)
// 7 SIM800L: connessione alla rete GSM
void aggiornaDisplay() {
    if (!oledPresente) return;

    DateTime ora = rtc.now();
    char buf[17];
    char tensione[8];

    oledRiga(0, formattaData(ora).c_str());

    oledRiga(1, F("Rete elettrica:"));
    oled.draw2x2String(0, 2, statoReteConfermato ? "ONLINE " : "OFFLINE");

    dtostrf(leggiTensioneBatteria(), 4, 2, tensione);
    snprintf(buf, sizeof(buf), "Batteria: %sV", tensione);
    oledRiga(4, buf);

    oledStatoArduino(true);

    oledRiga(6, sim800Attivo ? F("SIM800L: ATTIVO") : F("SIM800L: ASSENTE"));

    switch (statoGSM) {
        case 1: // registrata sulla rete di casa
        case 5: // registrata in roaming
            oledRiga(7, F("GSM: CONNESSO"));
            break;
        case 2:
            oledRiga(7, F("GSM: RICERCA..."));
            break;
        case STATO_GSM_NESSUNA_RISPOSTA:
            oledRiga(7, F("GSM: --"));
            break;
        default: // 0 = non registrata, 3 = negata
            oledRiga(7, F("GSM: NO RETE"));
            break;
    }
}

// ----- SIM800L: INIT -----

void inizializzaSIM800() {
    sim800Attivo = inviaComandoAT("AT", 1000);
    configuraSIM800();
}

// Configura il modulo: SMS in modalita' testo e SMS in arrivo lasciati in
// memoria SIM (li leggiamo a polling, compatibile con lo sleep, a differenza
// della modalita' URC "al volo"). Al power-up il modulo riparte in modalita'
// PDU, quindi va rifatta anche quando compare dopo l'avvio di Arduino o si
// riavvia.
void configuraSIM800() {
    sim800.println(F("AT+CMGF=1"));
    Serial.print(leggiRispostaSIM800(1000));
    sim800.println(F("AT+CNMI=2,0,0,0,0"));
    Serial.print(leggiRispostaSIM800(1000));
}

// Invia un comando AT e ne mostra la risposta sul monitor seriale.
// Restituisce true se il modulo ha risposto "OK".
bool inviaComandoAT(const char* comando, unsigned long attesaMs) {
    sim800.println(comando);
    bool ok = false;
    char precedente = 0;
    unsigned long inizio = millis();
    while (millis() - inizio < attesaMs) {
        if (sim800.available()) {
            char c = sim800.read();
            Serial.write(c);
            if (precedente == 'O' && c == 'K') ok = true;
            precedente = c;
        }
    }
    return ok;
}

```

```

    }
}
return ok;
}

// ----- SIM800L: STATO RETE GSM -----

// Chiede al modulo se e' registrato sulla rete GSM (AT+CREG?) e restituisce
// il campo "stat" della risposta "+CREG: <n>,<stat>", oppure
// STATO_GSM_NESSUNA_RISPOSTA se il modulo non risponde. Si ferma appena
// arriva "OK", senza aspettare il timeout.
uint8_t leggiRegistrazioneGSM() {
    char buf[48];
    uint8_t n = 0;
    buf[0] = '\0';

    sim800.println(F("AT+CREG?"));
    unsigned long inizio = millis();
    while (millis() - inizio < 1000 && n < sizeof(buf) - 1) {
        if (sim800.available()) {
            buf[n++] = sim800.read();
            buf[n] = '\0';
            if (strstr(buf, "OK") != NULL) break;
        }
    }

    const char* p = strstr(buf, "+CREG:");
    if (p == NULL) return STATO_GSM_NESSUNA_RISPOSTA;
    p = strchr(p, ',');
    if (p == NULL) return STATO_GSM_NESSUNA_RISPOSTA;
    return (uint8_t)atoi(p + 1);
}

void aggiornaStatoGSM() {
    // Se il modulo non risponde ai comandi AT e' inutile insistere
    statoGSM = sim800Attivo ? leggiRegistrazioneGSM() : STATO_GSM_NESSUNA_RISPOSTA;
}

// ----- SIM800L: INVIO SMS -----

void inviaSMSTutti(const String& testo) {
    // Modalita' testo: se il modulo si e' riavviato l'ha persa e senza questa
    // gli SMS non partono.
    sim800.println(F("AT+CMGF=1"));
    leggiRispostaSIM800(1000);

    inviaSMS(NUMERO_ALLARME_1, testo);
    delay(2000); // pausa tra un invio e l'altro
    inviaSMS(NUMERO_ALLARME_2, testo);
}

void inviaSMS(const char* numero, const String& testo) {
    sim800.print("AT+CMGS=\"");
    sim800.print(numero);
    sim800.println("\");
    delay(500);
    sim800.print(testo);
    delay(500);
    sim800.write(26); // Ctrl+Z
    delay(3000);
    while (sim800.available()) sim800.read();
}

// ----- SIM800L: SMS IN ARRIVO (a polling) -----

void controllaSMSNonLetti() {
    sim800.println(F("AT+CMGL=\"REC UNREAD\""));
    String risposta = leggiRispostaSIM800(3000);

    // Il modulo e' vivo se risponde con OK oppure con un errore: ERROR e' la
    // risposta a questo comando quando il modulo e' ancora in modalita' PDU
    // (com'e' dopo l'accensione) o non ha la SIM. Contare solo l'OK farebbe
    // risultare "assente" un modulo comparso dopo l'avvio di Arduino.

```

```

uint8_t esito = esitoRisposta(risposta);
sim800Attivo = (esito != RISPOSTA_INCOMPLETA);
if (!sim800Attivo) {
    // Se arrivano caratteri senza senso la velocita' seriale non coincide
    if (risposta.length() > 0) {
        Serial.print(F("SIM800L: risposta non valida: "));
        Serial.println(risposta);
    }
    return; // niente da leggere ne' da cancellare
}

if (esito == RISPOSTA_ERRORE) {
    // Il modulo e' comparso dopo l'avvio di Arduino, o si e' riavviato:
    // va riconfigurato. Si riprova con la lettura al prossimo ciclo.
    configuraSIM800();
    return;
}

if (risposta.indexOf("STAT0") != -1) {
    rispondiConStato();
}

// Pulisce la memoria SIM (letti e non letti) per evitare di riempirla
sim800.println(F("AT+CMGD=1,4"));
leggiRispostaSIM800(1000);
}

// Legge la risposta del SIM800L e torna appena e' completa, senza aspettare
// il timeout (che resta come limite massimo). E' completa quando termina con
// "OK" o "ERROR" e il modulo ha smesso di trasmettere. Se non arriva nulla,
// nemmeno l'eco del comando, si rinuncia dopo ATTESA_PRIMA_RISPOSTA_MS.
String leggiRispostaSIM800(unsigned long timeoutMs) {
    const unsigned long ATTESA_PRIMA_RISPOSTA_MS = 500;
    const unsigned long PAUSA_FINE_RISPOSTA_MS = 100;

    String buffer = "";
    unsigned long inizio = millis();
    unsigned long ultimoCarattere = inizio;
    while (millis() - inizio < timeoutMs) {
        while (sim800.available()) {
            buffer += (char)sim800.read();
            ultimoCarattere = millis();
        }
        if (buffer.length() == 0) {
            if (millis() - inizio >= ATTESA_PRIMA_RISPOSTA_MS) break;
        } else if (millis() - ultimoCarattere >= PAUSA_FINE_RISPOSTA_MS &&
            esitoRisposta(buffer) != RISPOSTA_INCOMPLETA) {
            break;
        }
    }
    return buffer;
}

// Guarda l'ultima riga della risposta: "OK" oppure una riga d'errore
// ("ERROR", "+CME ERROR: ...", "+CMS ERROR: ...") sono il codice finale; se la
// risposta non ne ha ancora uno (o non termina con un a-capo) e' incompleta.
uint8_t esitoRisposta(const String& s) {
    const char* c = s.c_str();
    int fine = s.length();
    if (fine == 0 || (c[fine - 1] != '\n' && c[fine - 1] != '\r')) {
        return RISPOSTA_INCOMPLETA;
    }
    while (fine > 0 && (c[fine - 1] == '\n' || c[fine - 1] == '\r')) fine--;

    int inizio = fine;
    while (inizio > 0 && c[inizio - 1] != '\n' && c[inizio - 1] != '\r') inizio--;

    if (fine - inizio == 2 && c[inizio] == 'O' && c[inizio + 1] == 'K') {
        return RISPOSTA_OK;
    }
    for (int i = inizio; i + 5 <= fine; i++) {
        if (strncmp(c + i, "ERROR", 5) == 0) return RISPOSTA_ERRORE;
    }
}

```

```

    return RISPOSTA_INCOMPLETA;
}

void rispondiConStato() {
    DateTime ora = rtc.now();
    String msg = "Stato sistema (" + formattaData(ora) + "):\n";
    msg += statoReteConfermato ? "Rete: presente\n" : "Rete: ASSENTE\n";
    msg += "Batteria: " + String(leggiTensioneBatteria(), 2) + "V";
    inviaSMSATutti(msg);
}

// ----- BATTERIA -----

void controllaBatteria() {
    float v = leggiTensioneBatteria();

    if (v < SOGLIA_BATTERIA_BASSA && !batteriaBassaSegnalata) {
        batteriaBassaSegnalata = true;
        DateTime ora = rtc.now();
        String msg = "Attenzione: batteria del sistema di monitoraggio scarica (";
        msg += String(v, 2) + "V) - " + formattaData(ora);
        inviaSMSATutti(msg);
    } else if (v > (SOGLIA_BATTERIA_BASSA + ISTERESI_BATTERIA)) {
        batteriaBassaSegnalata = false;
    }
}

// Lettura media di A0 (con AVcc come riferimento dell'ADC)
float leggiAdcBatteria() {
    analogRead(PIN_BATTERIA); // la prima lettura dopo un cambio di canale si scarta
    uint32_t somma = 0;
    for (uint8_t i = 0; i < LETTURE_ADC; i++) {
        somma += analogRead(PIN_BATTERIA);
    }
    return somma / (float)LETTURE_ADC;
}

// Lettura media del riferimento interno da 1.1V, sempre con AVcc come
// riferimento dell'ADC: piu' Vcc e' bassa, piu' il valore letto e' alto
// (5.0V -> circa 225, 4.0V -> circa 281).
float leggiAdcBandgap() {
    // AVcc come riferimento, canale interno 1.1V (MUX3..0 = 1110).
    // analogRead() reimposta ADMUX ad ogni chiamata, quindi non serve ripristinarlo.
    ADMUX = _BV(REFS0) | _BV(MUX3) | _BV(MUX2) | _BV(MUX1);
    delay(2); // il bandgap impiega circa 1 ms a stabilizzarsi dopo la selezione

    uint32_t somma = 0;
    for (uint8_t i = 0; i <= LETTURE_ADC; i++) {
        ADCSRA |= _BV(ADSC);
        while (ADCSRA & _BV(ADSC)) { }
        if (i > 0) somma += ADC; // la prima conversione si scarta
    }
    return somma / (float)LETTURE_ADC;
}

// Tensione di batteria in volt. Le due letture usano lo stesso riferimento
// (AVcc), quindi la Vcc si semplifica:
// V_A0 = 1.1 * lettura_A0 / lettura_bandgap
// e il risultato non dipende da quanto vale l'alimentazione di Arduino.
float leggiTensioneBatteria() {
    float lettBatteria = leggiAdcBatteria();
    float lettBandgap = leggiAdcBandgap();
    float vA0 = VREF_INTERNO * lettBatteria / lettBandgap;
    return vA0 * (R_ALTO + R_BASSO) / R_BASSO;
}

```