Indice generale

Guida per lo sviluppo software

Macchina a stati finiti

L'intero software è stato riscritto e sviluppato secondo il modello della macchina a stati finiti ed in particolare, secondo un automa di Mealy in cui la transizione da uno stato ad un altro dipende dallo stato attuale e da eventi esterni. Tale approccio consente di specificare con chiarezza le transizioni da eseguire tra gli stati in base agli eventi ed evitare stati di incoerenza causanti il malfunzionamento o il blocco del sistema.
Ogni sequenza di operazioni è chiaramente identificata e modellata attraverso stati ben precisi. L'implementazione dell'intero automa è realizzata attraverso una variabile indicante lo stato corrente, aggiornata ad ogni passaggio di stato. La scelta dell'esecuzione di un particolare stato avviene per merito di un costrutto switch il quale, ciclo dopo ciclo, processando la variabile indicante lo stato corrente, eseguirà il codice relativo. Tale codice è implementato attraverso l'uso di funzioni non bloccanti: tale approccio consente uno pseudo parallelismo del codice, evitando di assegnare il microcontrollore ad una specifica esecuzione per un periodo di tempo eccessivo tale da penalizzare l'esecuzione di altre funzioni.

La scrittura del software mediante le regole appena descritte consentono un'assoluta modularità ed una rapida scalabilità dello stesso con l'aggiunta di funzionalità che in seguito potrebbero essere richieste.
Inoltre, tale approccio è in piena sintonia con la filosofia del progetto Stima, rendendo l'intero software facilmente comprensibile a chiunque abbia voglia di realizzare la propria stazione meteorologica, in accordo all'idea che sta alla base dell'open source e delle specifiche tecniche RMAP.

Implementazione software della macchina a stati finiti

Ogni task del sistema è composto da:

1. Metodo implementativo delle funzionalità
2. Variabile di stato
3. Variabile booleana indicante se il task è in esecuzione

Per implementare un ipotetico task di esempio, è necessario creare:

1. Una variabile globale booleana indicante se il task è in esecuzione:

```
bool is_event_esempio = false;
```

2. Un nuovo tipo di variabile definendo i vari stati necessari ad implementare le funzionalità del task, come enumerazioni:

```
typedef enum {
```

```
    ESEMPIO_INIT,
    .
    ESEMPIO_END,
    ESEMPIO_WAIT_STATE
} esempio_state_t;
```

3. Una variabile globale del tipo appena definito:

```
esempio_state_t esempio_state = ESEMPIO_INIT;
```

4. La funzione implementante il task:

```
void esempio_task () {
  static esempio_state_t state_after_wait;
  static uint32_t delay_ms;
  static uint32_t start_time_ms;

  switch (esempio_state) {
    case ESEMPIO_INIT:
      state_after_wait = ESEMPIO_INIT;
      esempio_state = "stato successivo";
    break;

    .
    .
    .

    case ESEMPIO_END:
      noInterrupts();
      is_event_esempio = false;
      ready_tasks_count--;
      interrupts();
      esempio_state = ESEMPIO_INIT;
    break;

    case ESEMPIO_WAIT_STATE:
      if (millis() - start_time_ms > delay_ms) {
        esempio_state = state_after_wait;
      }
    break;
  }
}
```

Se nel corso dell'esecuzione del task è necessario attendere un certo intervallo di tempo attraverso lo stato di attesa non bloccante è possibile farlo mediante:

```
delay_ms = 10;
start_time_ms = millis();
state_after_wait = "stato successivo allo scadere del timeout di 10 ms";
esempio_state = ESEMPIO_WAIT_STATE;
```

La chiamata al task viene fatta nel loop() e implementata mediante la forma:

```
if (is_event_esempio) {
        esempio_task();
        wdt_reset();
}
```

Per attivare il task in un punto qualsiasi del codice, è possibile adottare la forma:

```
if (!is_event_esempio) {
        noInterrupts();
        is_event_esempio = true;
        ready_tasks_count++;
        interrupts();
}
```

SensorDriver

SensorDriver è la libreria scritta in C++ OOP che implementa la lettura dei sensori attraverso interfacce standard su bus I^2C.
Per la lettura dei sensori, viene creato un array del tipo SensorDriver *sensors[COUNT] a cui ad ogni elemento dell'array corrisponde un oggetto di tipo SensorDriver che implementa i metodi descritti nel seguito.

- SensorDriver(const char* driver, const char* type)
  - Costruttore
  - const char* driver: stringa di 3 caratteri contenente il nome del driver
  - const char* type: stringa di 3 caratteri contenente il nome del sensore

- virtual void setup(const uint8_t address, const uint8_t node = 0)
  - operazioni di inizializzazione del sensore
  - const uint8_t address: indirizzo I^2C del sensore
  - const uint8_t node: nodo all'interno della rete

- virtual void prepare(bool is_test = false)
  - inizializzazione del sensore precedente alla lettura
  - bool is_test: se false il sensore viene preparato per effettuare le normali procedura di lettura, se true il sensore si predispone per leggere valori particolari "di test" utili alla verifica di funzionamento dello stesso

- virtual void get(int32_t *values, uint8_t length)
  - lettura dei valori dal sensore in formato numerico intero a 32 bit con segno
  - int32_t *values: puntatore all'array di ritorno dei valori
  - uint8_t length: numero di valori da leggere dal sensore

- virtual void getJson(int32_t *values, uint8_t length, char *json_buffer, size_t json_buffer_length = JSON_BUFFER_LENGTH)
  - lettura dei valori dal sensore in formato JSON
  - int32_t *values: puntatore all'array di ritorno dei valori
  - uint8_t length: numero di valori da leggere dal sensore
  - char *json_buffer: buffer di ritorno della stringa contente il JSON
  - size_t json_buffer_length: lunghezza del buffer

- static SensorDriver *create(const char* driver, const char* type)
  - crea un'istanza di SensorDriver per un sensore specifico
  - const char* driver: stringa di 3 caratteri contenente il nome del driver
  - const char* type: stringa di 3 caratteri contenente il nome del sensore

- static void createAndSetup(const char* driver, const char* type, const uint8_t address, const uint8_t node, SensorDriver *sensors[], uint8_t *sensors_count)
  - richiama in sequenza i metodi create e setup assegnando la nuova istanza del sensore all'array delle istanze dei sensori incrementandone la variabile corrispondente che ne indica la dimensione
  - const char* driver: stringa di 3 caratteri contenente il nome del driver
  - const char* type: stringa di 3 caratteri contenente il nome del sensore
  - const uint8_t address: indirizzo I^2C del sensore

    - int8_t node: nodo all'interno della rete
    - const u SensorDriver *sensors[]: array delle istanze dei sensori
    - uint8_t *sensors_count: numero di istanze create

- char *getDriver()
    - ritorna il puntatore alla stringa contente il driver del sensore

- char *getType()
    - ritorna il puntatore alla stringa contente il tipo del sensore

- uint8_t getAddress()
    - ritorna l'indirizzo I^2C del sensore

- uint8_t getNode()
    - ritorna il nodo del sensore

- uint32_t_t getStartTime()
    - ritorna il valore in millisecondi relativo all'istante iniziale in cui viene richiesto il delay

- uint32_t_t getDelay()
    - ritorna il valore in millisecondi indicante l'attesa richiesta

- bool isEnd()
    - ritorna true quando la procedura get del sensore è terminata, false se la procedura è in corso

- bool isSuccess()
    - ritorna true se la procedura get termina con successo, false in caso contrario

- bool isSetted()
    - ritorna true se l'operazione setup è stata eseguita con successo, false in caso contrario

- bool isPrepared()
    - ritorna true se l'operazione prepare è stata eseguita con successo, false in caso contrario

- void resetPrepared()
    - resetta il flag indicante la corretta esecuzione della procedura prepare (flag ritornato dalla procedura isPrepared())

Microcontrollori e hardware in modalità risparmio energetico

Per garantire il funzionamento della stazione con batteria e pannello fotovoltaico, i microcontrollori sono impostati in modalità a basso consumo. Tale modalità è raggiunta con lo spegnimento fisico di tutta la strumentazione non strettamente necessaria che sarà alimentata solo nel momento in cui risulti utile (ad esempio: il modulo GSM/GPRS ed alcune periferiche dei microprocessori).

In particolare i moduli Stima Ethernet o Stima GSM/GPRS sono posti in modalità power down e risvegliati con interrupt dell'RTC con cadenza del secondo.
Analogamente, il modulo Stima I2C-Rain è risvegliato dall'interrupt dovuto ad una basculata del pluviometro e il modulo Stima I2C-TH viene svegliato tramite interrupt del timer interno.
Entrambi i moduli Stima I2C-Rain e Stima I2C-TH possono essere risvegliati attraverso matching dell'indirizzo I^2C del microcontrollore. Ciò consente di porre tutta la strumentazione in modalità risparmio energetico e qualora un qualsiasi dispositivo multi-master sul bus, si risvegli autonomamente o in seguito ad un evento esterno (esempio: segnalazione di pioggia dal pluviometro), potrà risvegliare tutti i moduli multi-master necessari, con un semplice indirizzamento I^2C del dispositivo specifico.

Salvataggio dei dati su SD-Card

Tutti i dati acquisiti e le relative ed eventuali elaborazioni effettuate, sono salvate su scheda SD-Card e conseguentemente inviati al server RMAP.
Per assolvere tali funzioni ed ottimizzare il funzionamento complessivo della stazione meteorologica in merito ad overhead del tempo di cpu per la ricerca dei file ed all'uso dello spazio sul disco, viene creato un file per ogni giorno di registrazione di dati, salvando all'interno tutti i dati dei sensori relativi a quel giorno.
Per gestire la modalità di invio dati al server, è presente un unico file in cui viene scritto di volta in volta, il puntatore corrispondente alla data ed ora dell'ultimo dato trasmesso al server RMAP. Per effettuare un nuovo trasferimento dei dati a partire da una specifica data antecedente a quella del puntatore ai dati correnti, è sufficiente un aggiornamento di tale puntatore con la data desiderata: sarà compito del software ricercare il primo dato utile successivo a tale data.
Nello specifico, ogni file di dati assume il nome nel formato aaaa_mm_gg.txt in cui:

- aaaa: anno con 4 cifre
- mm: mese con 2 cifre
- gg: giorno con 2 cifre

In ogni file, ogni riga corrisponde ad un dato di un sensore ed in particolare, ogni riga è della lunghezza di MQTT_SENSOR_TOPIC_LENGTH + MQTT_MESSAGE_LENGTH bytes. Tali valori sono delle #define situate nel file mqtt_config.h nella cartella arduino/sketchbook/libraries/RmapConfig.
Ogni riga è salvata nel formato: TRANGE/LEVEL/VAR {"v": VALUE, "t": TIME}
Il file contenente il puntatore all'ultimo dato trasmetto assume il nome mqtt_ptr.txt e contiene un dato binario della dimensione di 4 bytes senza segno corrispondente al numero di secondi dal 00:00:00 del 01/01/1970 indicante la data ed ora dell'ultimo dato trasmetto attraverso MQTT.