UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

DIPARTIMENTO DI SCIENZE TEORICHE E APPLICATE
CORSO DI STUDIO TRIENNALE IN INFORMATICA - F004

Sviluppo di un sistema embedded per il controllo della temperatura in una camera di collaudo

Relatore:
Prof. Carlo Dossi

Tesi di Laurea di:
Mattia Papaccioli - 747053

Tutor Aziendale:
Edoardo Scaglia

Azienda ospitante:
AMEL SRL

Anno Accademico:
2025/2026

Indice

1

Introduzione

1.1 Abstract

La camera di collaudo, situata presso AMEL s.r.l., è utilizzata per testare carichi resistivi. Questi producono calore, rendendo necessario il raffreddamento dell'ambiente, soprattutto nei mesi più caldi.

Il sistema embedded controlla la frequenza di una ventola di raffreddamento mediante un controllo PID:

1. La temperatura ambiente viene letta tramite un sensore.
2. Questa viene utilizzata come ingresso del sistema PID.
3. L'algoritmo PID calcola la tensione da inviare a un inverter, il quale

determina la frequenza della ventola in base all'errore attuale (azione proporzionale) e a quello accumulato (azione integrativa).

1. Questo sistema di retroazione negativa viene applicato continuamente per

mantenere costante la temperatura ambiente.

La comunicazione tra l'inverter e il sistema embedded avviene tramite il protocollo MODBUS RTU [1].

Il sistema embedded fornisce inoltre un'interfaccia per regolare la temperatura target dell'ambiente mediante un display touchscreen. La GUI è sviluppata utilizzando la libreria LVGL [2].

Il sistema embedded è collegato alla rete aziendale tramite Ethernet; attraverso un web server sarà possibile regolare la temperatura target anche da remoto.

La comunicazione tra il sistema embedded e il web server avviene mediante scrittura su file o IPC (Inter-Process Communication).

È ancora da valutare l'impiego di un database per registrare la temperatura nel tempo. In tal caso, il web server gestirà l'interazione con esso.

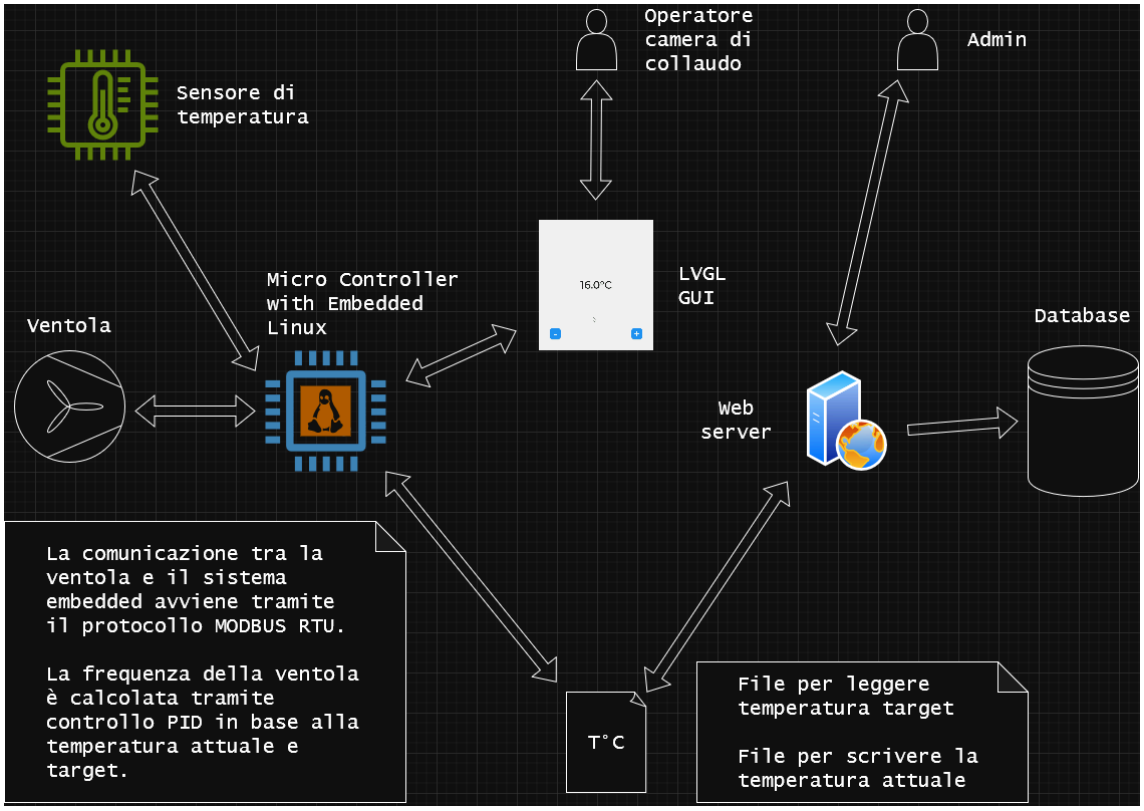

Figura 1 — Diagramma che illustra il sistema

1.2 Struttura del codice sorgente

Il codice sorgente del progetto è pubblicato su GitHub [3]. Il codice sorgente è composto da più moduli:

- `common-control`: contiene funzioni helper comuni di aiuto e definizioni per la comunicazione tra interfaccia grafica e backend.
- `temp-control`: responsabile per la GUI scritta in LVGL.
- `pid-control`: responsabile per il controllo PID, la rilevazione della temperatura e la comunicazione MODBUS RTU [1].

È stato intrapreso questo approccio per garantire modularità del codice, consentendo in futuro di poter rimpiazzare ciascun modulo con relativa facilità, rispetto a un approccio monolitico.

2

Moduli

2.1 `common-control`

Il modulo `common-control` contiene funzioni helper comuni, header di configurazione globale e script per inizializzare e avviare l'intero sistema.

Viene utilizzata la cartella di sistema `/opt/amel/` per contenere gli script e i file di tutti i moduli.

2.1.1 Comunicazione tra GUI e PID

Per fare in modo che i moduli `temp-control` e `pid-control` comunichino tra di loro è stato necessario sviluppare un sistema di segnali e scrittura e lettura su file.

Quando un operatore cambia la temperatura target dall'interfaccia sul display LCD essa viene scritta sul file `/opt/amel/target-temperature`.

Analogamente, il processo PID quando rileva una temperatura tramite i sensori DS18B20 [4], scrive quest'ultima sul file `/opt/amel/current-temperature/sX`, con x che rappresenta il numero del sensore sul bus. Subito dopo aver scritto, viene mandato un segnale a `temp-control`, che a sua volta legge il file e aggiorna la temperatura del sensore spiegata successivamente.

2.1.2 `logging.h`

Per stampare a schermo in modo ordinato i messaggi dell'applicazione è stato implementata una semplice libreria in C che consente di loggare, anche con stringhe formattate, ai livelli `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR` e `FATAL`. Si può decidere a che livello filtrare i messaggi e anche se scrivere su file nel syslog o sulla console.

Per evitare che più log si sovrascrivino a vicenda viene utilizzato un meccanismo di mutex. È stata aggiunta un'opzione per decidere la precisione del timer per intervalli di tempo sotto al secondo, utile per debuggare la regolarità del controllo PID.

2.1.3 `bash` templating with c preprocessor

Per evitare duplicazioni di costanti all'interno del modulo, è stato utilizzato il preprocessore del linguaggio c in modo creativo. Avendo definito le configurazioni in `include/config.h`, sono state utilizzate come input per dei template dal quale si ricavano gli script di inizializzazione e di avvio dell'intero progetto, rispettivamente `init.sh` e `run.sh`. Facendo compilare `gcc` con la flag `-E` possiamo sfruttare il preprocessore senza effettuare compilazione, assemblaggio e linking.

In questo modo e necessario cambiare le variabili solamente nell'header di configurazione per averle aggiornate anche negli script.

2.1.4 Esecuzione del modulo

La prima volta che si avvia il dispositivo embedded e necessario eseguire lo script `/opt/amel/init.sh`, che si occupa di inizializzare le directory per i sensori di temperatura e setuppare l access control. Per avviare front e back end in un unico comando si utilizza lo script `/opt/amel/run.sh`. Esso avvia per primo il main di `temp-control`, che si mette in ascolto del segnale dal `pid-control`. Successivamente parte `pid-control` che inizializza il bus one-wire e scrive il numero di sensori nel file `/opt/amel/number-sensors`. Una volta fatto cio, manda il segnale a `temp-control` e i due main iniziano l'esecuzione, comunicando update in temperatura target e temperatura attuale tramite segnali con il comando `kill`.

2.1.5 Web server CGI

Per monitorare e controllare il sistema da remoto è stato implementato un semplice web server CGI che mostra la temperatura attuale e consente di modificare la temperatura target tramite due bottoni. Il server web è ospitato direttamente sul dispositivo embedded e comunica con i moduli `temp-control` e `pid-control` leggendo e scrivendo sui file di interfaccia.

2.2 `temp-control`

Per controllare la temperatura della camera di collaudo, l'operatore imposta la temperatura target mediante un display touchscreen LCD «NOME DISPLAY». L'interfaccia grafica è sviluppata utilizzando la libreria LVGL [2] e si è utilizzato un template [5] contenente il porting su Linux fornito dagli sviluppatori della libreria.

2.2.1 Schermata principale

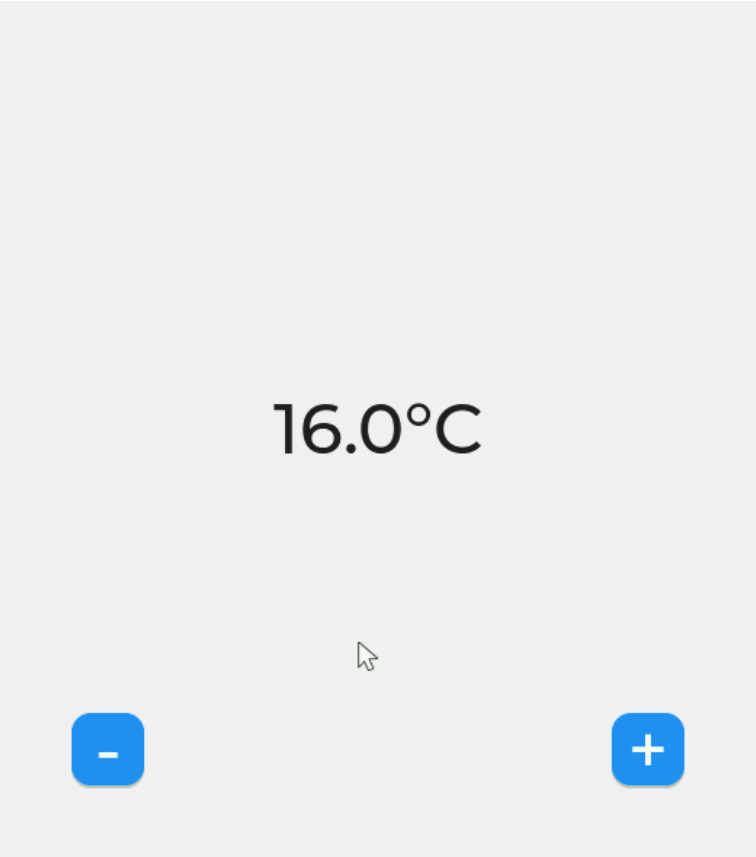

Figura 2 — Interfaccia grafica per il controllo della temperatura

Nell'interfaccia vengono mostrate temperatura target, temperatura attuale dei sensori collegati sul bus one-wire e due bottoni per aumentare e diminuire la temperatura target.

```c
void set_target_temperature(float t)
{
    LOG_DEBUG("debug callback -> %.1f\n", target_temperature);
    target_temperature += t;
    lv_label_set_text_fmt(target_temperature_label,
    target_temperature_format, target_temperature);
    write_float_to_file(TARGET_TEMPERATURE_FILE, target_temperature);
    kill(PID_control_PID, SIGUSR1);
}

static void increment_temperature(lv_event_t * e)
{
    if(lv_event_get_code(e) != LV_EVENT_CLICKED) {
        return;
    }
    set_target_temperature(1);
}

static void decrement_temperature(lv_event_t * e)
{
    if(lv_event_get_code(e) != LV_EVENT_CLICKED) {
        return;
    }
    set_target_temperature(-1);
}
```

Codice 1 — Funzioni di callback per i bottoni di incremento e decremento della temperatura target

Quando viene premuto il pulsante per aumentare o diminuire la temperatura target, viene invocata una funzione di callback che si occupa di aggiornare la `label` con la temperatura target sullo schermo e di scriverla nell'apposito file.

2.2.2 Backend I/O

I backend utilizzati da LVGL per l'I/O sono libevdev e il framebuffer device. Sono stati scelti per la loro semplicità e il ridotto utilizzo di risorse.

Libevdev [6] è una libreria che gestisce gli eventi di input: riceve i tocchi dal touchscreen e li passa all'interfaccia grafica.

Il framebuffer device è semplicemente il file `/dev/fb0`, scritto dalla GUI, che contiene il colore di ciascun pixel dello schermo.

2.2.3 Compilazione della GUI

Per la compilazione dell'applicazione è necessaria una toolchain adatta all'architettura ARM. Nel nostro caso, ci affidiamo al compilatore e alle librerie fornite da Buildroot.

`cross_compile_setup.cmake`

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(tools ~/buildroot/output/host/bin/arm-buildroot-linux-gnueabihf-)
set(CMAKE_C_COMPILER ${tools}gcc)
set(CMAKE_CXX_COMPILER ${tools}g++)

set(EVDEV_INCLUDE_DIRS ~/buildroot/output/staging/usr/include/libevdev/)
set(EVDEV_LIBRARIES ~/buildroot/output/staging/usr/lib/libevdev.so)

set(BUILD_SHARED_LIBS ON)
```

Codice 2 — cross_compile_setup.cmake

Il comando `cmake -DCMAKE_TOOLCHAIN_FILE=./cross_compile_setup.cmake -B build -S .` genera i Makefile necessari per la cross-compilazione, che vengono poi eseguiti con `make -C build -j` [7].

LVGL viene compilata come libreria condivisa, mentre l'applicazione come eseguibile.

2.2.4 Branches

Per organizzare efficientemente la repository sorgente, è stato realizzato un branching, creando una repository per lo sviluppo ed una per il dispositivo target. Esse sono uguali completamente tranne per il file `lv_conf.h`.

Per la branch di sviluppo, esso usa come backend `x11` e contiene dei sanity check, utili in sviluppo ma limitanti in termini di performance.

Per la branch del dispositivo target sono stati disabilitati i sanity checks e utilizzata come backend il device `/dev/fb0`.

Per proteggere il file `lv_conf.h` è stato aggiunto un file `.gitattributes` contenente `lv_conf.h merge=ours`.

Questo speciale file di git, comunica al version control system che durante il merge delle branch di mantenere il file come si trova nella branch da cui si sta effettuando il merge, consentendo di mantenere separate le due configurazioni senza preoccuparsi di sovrascriverle accidentalmente.

2.3 `pid-control`

2.3.1 Sensore di temperatura

I sensori di temperatura utilizzati sono due DS18B20 [4] collegati in parallelo su un bus 1-Wire.

Il microcontrollore si comporta da master sul bus e richiede periodicamente la temperatura ai sensori.

Il binario `PID` legge periodicamente e calcola il valore di output del controller PID in base alla temperatura misurata e al setpoint desiderato.

Inizialmente esso conta il numero di sensori sul bus, alloca la memoria necessaria per immagazzinare gli uuid dei sensori e poi legge effettivamente quest'ultimi in memoria.

È stato preferito questo approccio per evitare complicazioni con `realloc` rispetto a leggere direttamente in un ciclo unico sia il numero di sensori che gli id. Questa procedura viene effettuata solamente una volta all'avvio e non ha un impatto significativo sulla performance dell'eseguibile.

Un approccio senza salvare gli id dei sensori porterebbe a una chiamata alla funzione `DS18X20_find_sensor` ripetutamente e sarebbe uno spreco di cicli di CPU quindi sacrifichiamo un po' di memoria per questo.

```c
typedef struct
{
    char id[16];
    int16_t temperature;
    uint8_t uint_id[OW_ROMCODE_SIZE];
} sensor;

void init_onewire_sensors()
{
    uint8_t diff = OW_SEARCH_FIRST;

    while (diff != OW_LAST_DEVICE)
    {
        sensors_count++;
        DS18X20_find_sensor(&diff, id);
    }

    write_char_to_file(NUMBER_OF_SENSORS_FILE, sensors_count);
    LOG_INFO("Found %d sensors on the 1-wire bus", sensors_count);

    sensors = malloc(sizeof(sensor) * sensors_count);

    diff = OW_SEARCH_FIRST;
    int i = 0;
    while (diff != OW_LAST_DEVICE)
    {
        DS18X20_find_sensor(&diff, id);
        sensor s;
        for (int i = 0; i < OW_ROMCODE_SIZE; i++)
        {
            s.uint_id[i] = id[i];
        }
        sprintf(s.id, "%02hx%02hx%02hx%02hx%02hx%02hx%02hx%02hx",
        id[0], id[1],
              id[2], id[3], id[4], id[5], id[6], id[7]);
        s.temperature = 0;
        sprintf(s.file, CURRENT_TEMPERATURE_FILE "/s%d", i);
        LOG_INFO("sensor %d id %s", i, s.id);
        sensors[i] = s;
        i++;
    }
}
```

Codice 3 — `pid/src/main.c`

CRC-8 Il controllo di integrità dei dati letti dai sensori DS18B20 viene effettuato mediante l'algoritmo CRC-8, implementato nella funzione `crc8`. Tale funzione utilizza il polinomio 0x18, corrispondente a (x^8 + x^5 + x^4 + x^0), per calcolare il checksum sui dati dello scratchpad. Questo permette di verificare che i dati ricevuti dal sensore non siano corrotti durante la trasmissione sul bus 1-Wire. In caso di errore CRC, la lettura viene scartata e può essere ritentata.

2.3.2 MODBUS RTU

Per comunicare con l'inverter che controlla la ventola di raffreddamento, è stato utilizzato il protocollo MODBUS RTU tramite l'apposita libreria `libmodbus` [1].

2.3.3 Controllo PID

Il controllo PID (Proporzionale, Integrale e Derivativo) è un sistema di retroazione negativa che permette di reagire a un errore rispetto a un valore target.

Esso viene utilizzato per mantenere la temperatura costante nella camera di collaudo. Prende in input la temperatura rilevata dai sensori e la temperatura desiderata all'interno della stanza e restituisce in output la tensione con la quale comunichiamo all'inverter la frequenza della ventola di raffreddamento.

Monotonic clock

Per campionare la temperatura nella stanza ad una frequenza costante, fondamentale per un corretto calcolo PID, è stato utilizzato l'header `<sys/timerfd.h>` della `Standard C library`. Queste chiamate di sistema creano e operano su un timer che consegna segnali di scadenza del timer ad intervalli regolari tramite un file descriptor.

Scheduler priority

È stata assegnata la massima priorità di scheduler al programma tramite l'header `<sched.h>` per evitare interrupt durante la misurazione e per cercare di mantenere più costante possibile la periodicità del campionamento.

2.3.4 Admin Control

2.3.5 Logging and Monitoring

Per monitorare e registrare l'andamento del controllo PID nel tempo, è stata implementata una funzionalità di logging su file CSV. All'avvio del programma, viene creato un file CSV con un nome basato sul timestamp corrente, contenente un'intestazione con le colonne per il tempo, la temperatura target, le temperature dei singoli sensori e l'output del PID. Ad ogni ciclo del controllo, vengono aggiunti i valori attuali al file, permettendo un'analisi successiva dei dati mediante strumenti di analisi o fogli di calcolo.

3

Embedded Linux

3.1 Hardware

La scheda utilizzata per il sistema embedded è sviluppata da AMEL e comprende:

- una host-board Ganador (rev. 4) che integra:
  - una memoria SD utilizzata come disco fisso;
  - un'interfaccia Ethernet;
  - un'interfaccia seriale;
  - un display touchscreen.
- un system-on-module Vulcano-A5 che integra:
  - una CPU Atmel ARM9 AT91SAM9X35 @ 400MHz;
  - 128 MB DDR2 SDRAM;
  - 256 MB NAND Flash;
  - SODIMM200 interface;
  - 10/100 Mbps Ethernet MAC Controller;
  - 2x USB 2.0 Host, 1x USB 2.0 Host/Device;
  - 3x USARTs;
  - TFT LCD Controller with TTL / LVDS support.

3.2 Costruzione del sistema

Per orchestrare il sistema è stato utilizzato Linux [8]. È stato creato un kernel personalizzato con soli i moduli essenziali per il funzionamento, data la limitatezza delle risorse hardware. Lo strumento utilizzato per completare l'opera è Buildroot [9], che consiste essenzialmente in una serie di Makefile per installare e cross-compilare tutte le librerie e i pacchetti necessari alla costruzione e all'esecuzione del sistema. Inoltre, si occupa di creare il filesystem e di prepararlo in un'immagine pronta per essere scritta sulla scheda SD del sistema embedded.

Per aggiungere l'interfaccia grafica sviluppata con LVGL è stato necessario creare un nuovo pacchetto in Buildroot.

All'avvio del sistema, il bootloader del chip attiva AT91bootstrap [10], che a sua volta avvia Barebox [11], il quale carica il kernel in memoria.

3.3 Personalizzazione di buildroot

Per aggiungere un pacchetto a Buildroot è necessario inserire una nuova voce nella cartella `package`, comprendente un file `Config.in` e un file `.mk`.

Questi due file contengono le istruzioni che consentono a Buildroot di risolvere le dipendenze, scaricare e installare il pacchetto nel filesystem del dispositivo target.

3.3.1 Il pacchetto `amel-common-control`

```
 AMEL_COMMON_CONTROL_VERSION = v0.9
 AMEL_COMMON_CONTROL_SITE =
 git@git.amelchem.com:mpapaccioli/common-control.git
 AMEL_COMMON_CONTROL_SITE_METHOD = git

 define AMEL_COMMON_CONTROL_BUILD_CMDS
$(TARGET_MAKE_ENV) TOOLS=$(TARGET_CROSS) STAGE=$(STAGING_DIR) \
cmake -S $(@D) -B $(@D)/build -D CMAKE_BUILD_TYPE=Release -D  \
CMAKE_INSTALL_PREFIX=/usr             \
-DCMAKE_TOOLCHAIN_FILE=$(@D)/user_cross_compile_setup.cmake   \
-DBUILD_SHARED_LIBS=ON
$(TARGET_MAKE_ENV) cmake --build $(@D)/build
make -C $(@D) source/init.sh source/run.sh
 endef

 define AMEL_COMMON_CONTROL_INSTALL_TARGET_CMDS
$(TARGET_MAKE_ENV) DESTDIR=$(TARGET_DIR) cmake --install $(@D)/build
mkdir -p $(STAGING_DIR)/usr/include/common-control
mkdir -p $(STAGING_DIR)/usr/lib

cp -r $(@D)/build/libcommon-control.so* $(STAGING_DIR)/usr/lib/
cp -r $(@D)/include/common-control.h
$(STAGING_DIR)/usr/include/common-control
cp -r $(@D)/include/logging.h $(STAGING_DIR)/usr/include/common-control

$(INSTALL) -d $(TARGET_DIR)/opt/amel
$(INSTALL) -m 0755 $(@D)/source/init.sh $(TARGET_DIR)/opt/amel/init.sh
$(INSTALL) -m 0755 $(@D)/source/run.sh $(TARGET_DIR)/opt/amel/run.sh
 endef

 $(eval $(generic-package))
```

Codice 4 — `amel-common-control.mk`

3.3.2 Il pacchetto `amel-temp-control`

```
 AMEL_TEMP_CONTROL_VERSION = 44c17c6f2c492f1f3c7d8a6767df390c8d13eb9c
 AMEL_TEMP_CONTROL_SITE = git@git.amelchem.com:mpapaccioli/temp-
control.git
 AMEL_TEMP_CONTROL_SITE_METHOD = git

 AMEL_TEMP_CONTROL_DEPENDENCIES = libevdev
 AMEL_TEMP_CONTROL_GIT_SUBMODULES = YES

 define AMEL_TEMP_CONTROL_BUILD_CMDS
cmake -DCMAKE_TOOLCHAIN_FILE=$(@D)/user_cross_compile_setup.cmake \
  -B $(@D)/build -S $(@D)
make -C $(@D)/build -j @cmake

 endef

 define AMEL_TEMP_CONTROL_INSTALL_TARGET_CMDS
$(INSTALL) -d $(TARGET_DIR)/opt/amel-temp-control/
cp $(@D)/build/bin/lvglsim $(TARGET_DIR)/opt/amel-temp-control/main
cp -r $(@D)/build/lvgl/lib/* $(TARGET_DIR)/usr/lib

 endef

 $(eval $(generic-package))
```

Codice 5 — amel-temp-control.mk

Inizialmente, viene clonata la repository temp-control, contenente la GUI, al commit specificato, inizializzando i sottomoduli e verificando la presenza della dipendenza `libevdev`.

Successivamente, vengono cross-compilate la libreria LVGL e l'applicazione con interfaccia grafica utilizzando il compilatore ARM fornito da Buildroot.

Infine, i binari della libreria e l'eseguibile dell'applicazione vengono installati sulla macchina target.

3.3.3 Il pacchetto `amel-pid`

```
    AMEL_PID_VERSION = v0.0.3
    AMEL_PID_SITE = git@git.amelchem.com:mpapaccioli/pid.git
    AMEL_PID_SITE_METHOD = git

    AMEL_PID_DEPENDENCIES = libmodbus

    define AMEL_PID_BUILD_CMDS
make -C $(@D) CC=$(TARGET_CC)
    endef

    define AMEL_PID_INSTALL_TARGET_CMDS
$(INSTALL) -d $(TARGET_DIR)/opt/amel-pid/

cp $(@D)/pid $(TARGET_DIR)/opt/amel-pid/pid

    endef

    $(eval $(generic-package))
```

Codice 6 — `amel-pid-control.mk`

Analogamente, è stato creato il pacchetto `amel-pid-control` per cross-compilare e installare la libreria PID sviluppata in C++.

3.3.4 Modifica pacchetti networking

È stata creata una rete virtuale ed è stato aggiunto un SSH server [12] per consentire il collegamento remoto al dispositivo embedded. È stato necessario modificare lo script in `/etc/init.d/S50network` per configurare l'interfaccia di rete virtuale `eth0` con un indirizzo IP statico all'avvio del sistema e permettere il login all'utente `root` tramite password modificando il file `/etc/ssh/sshd_config`. Dopo aver ripetuto questo processo per tutti i pacchetti desiderati, il filesystem e l'immagine del kernel vengono assemblati in un file `sdcard.img` pronto per essere scritto su una scheda SD e avviato sul dispositivo embedded. Inoltre, la rete virtuale è stata creata in modo da condividere l'accesso a Internet e dopo aver impostato come default gateway il PC creatore della rete bridge è possibile accedere alla rete globale dal sistema embedded. È stato utilizzato il DNS server di Google `8.8.8.8`. L'accesso a Internet è necessario per la configurazione di NTP [13], come descritto in seguito.

3.3.5 Installazione del modulo c210x per la porta seriale

Per consentire la comunicazione seriale tramite la porta RS-232 della host-board Ganador, è stato necessario includere il modulo kernel `c210x` per il controller USB-seriale. Il modulo è stato aggiunto tramite il menu di configurazione di Buildroot, selezionandolo in `Kernel modules -> USB Serial Converter support -> USB CP210x family of UART Bridge Controllers` dal comando `make linux-menuconfig`.

3.3.6 Il pacchetto NTP

Per garantire la corretta sincronizzazione dell'orologio di sistema è stato necessario installare il client `ntpd` per il Network Time Protocol tramite Buildroot.

3.4 Custom patches a `barebox` e `linux`

A causa dei rebuild frequenti in fase di sviluppo, sono stati patchati i codici sorgenti di `barebox` [11] e `linux` [8].

È stata effettuata una clonazione della repository del kernel Linux di Amel ed è stato abilitato di default il modulo cp210x, modificando il file `/drivers/usb/serial/Kconfig`.

Analogamente, è stato cambiato l'ordine nel bootloader Barebox, prioritarizzando il device `mmc2` invece che `nand`.

4

Installazione del sistema

4.1 Comunicazione con l'inverter

Per comunicare con l'inverter via modbus e necessario configurarlo per accettare comandi da remoto tramite il display integrato. Dopodiche, con modus, si scrive nel registro della word di controllo il comando di start che corrisponde in esadecimale al valore 0x047f. Il criptico manuale dell'inverter e stato difficile da decifrare ma dopo varie riletture e tentativi siamo riusciti a farlo partire.

4.2 Regolazione dei parametri del controllo PID

Per regolare i parametri del controllo PID e stato intrapreso un approccio empirico. E stato necessario riscaldare la camera con un carico resistivo fino a 50 gradi Celsius. Abbiamo utilizzato il sensore DS18B20 menzionato in precedenza per misurare la temperatura digitalmente da utilizzare nel calcolo del PID ed un termistore gia presente nella camera come riferimento. Come previsto vi e stata un leggera incongruenza tra i due sensori. Dopodiche abbiamo iniziato a provare vari valori per il parametro Proporzionale, fino al punto in cui il sistema si e avvicinato al setpoint, nel nostro caso di 40 gradi Celsius ed ha iniziato a mantenere una temperatura stabile, di circa 43 gradi Celsius. Nel nostro caso visto che stiamo raffreddando, abbiamo utilizzato dei parametri con segno negativo. La differenza di magnitudine tra il parametro P (-3000) ed (-15) I e considerevole.

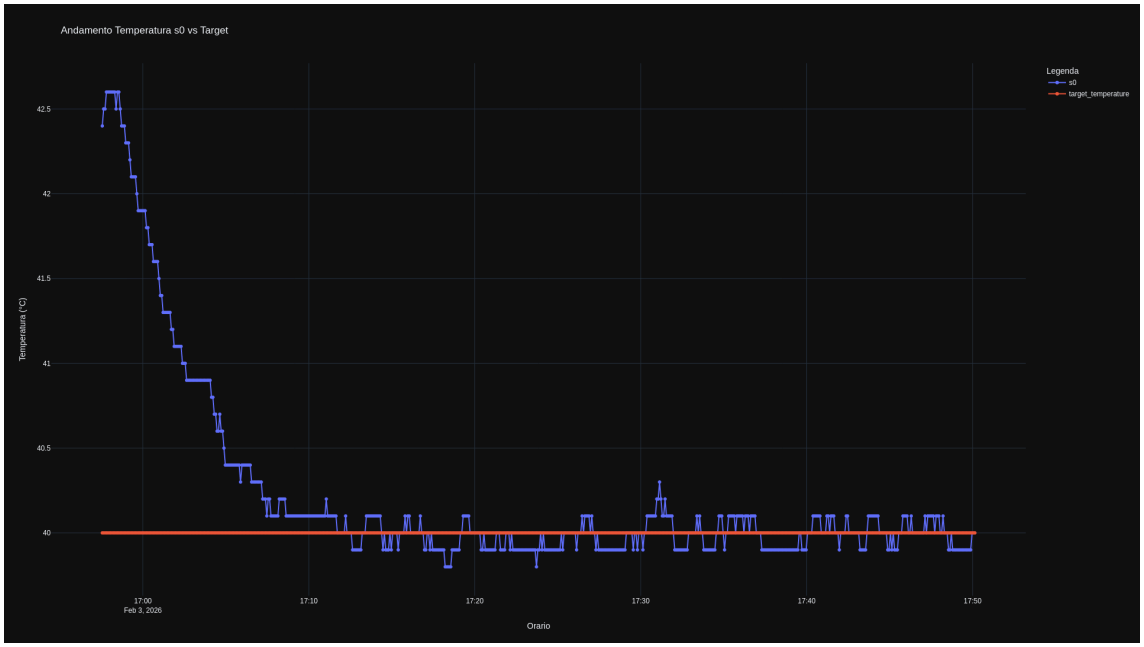

Figura 3 — Andamento della temperatura

Successivamente abbiamo iniziato ad aumentare il parametro Integrale per approcciare piu dolcemente ed accuratamente il setpoint. Una volta arrivati al setpoint il contributo della parte proporzionale diventa nullo e il componente integrale si occupa di mantenere il sistema stabile, come visibile nel seguente grafico.

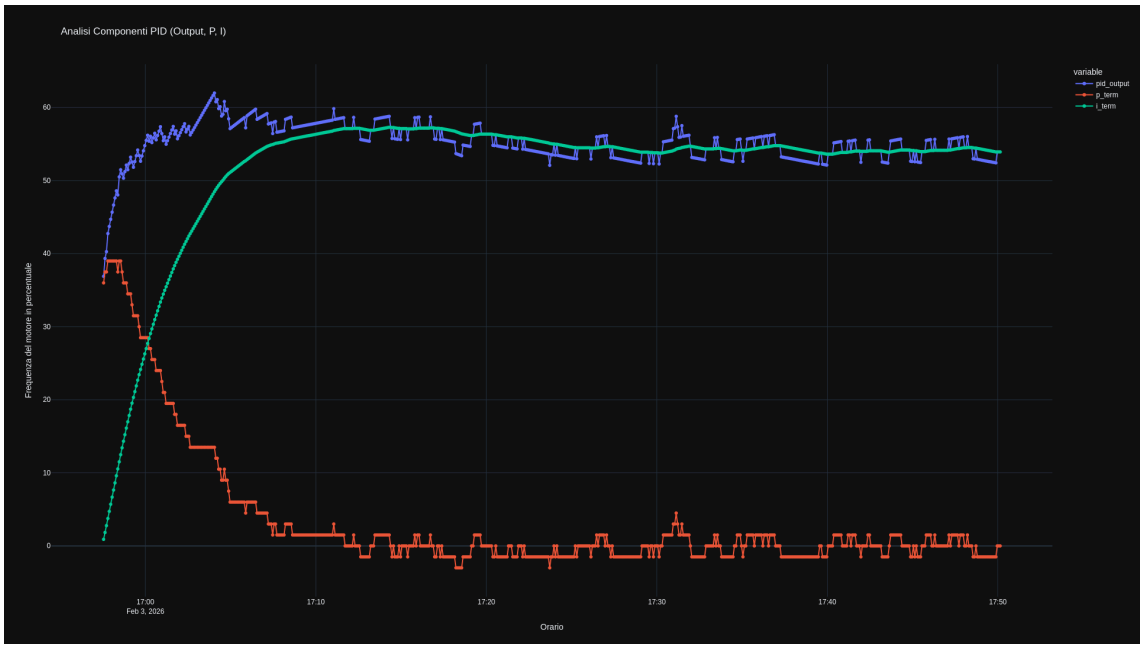

Figura 4 — Contributo dei vari parametri del controllo pid

4.3 Risoluzione di bug

Durante l'installazione del sistema, come prevedibile, abbiamo incontrato vari bug e trovato di conseguenza delle ottimizzazioni impensate durante la fase di sviluppo.

4.3.1 `lvgl` e single threaded

Per esempio, ho modificato il callback per aggiornare la temperature nel modulo `temp-control`: inizialmente anch'esso veniva aggiornamìto tramite il segnale del eseguibile del `pid-control` ma dopo aver riscontrato un bug relativo alla natura single threaded di `lvgl`, ho cambiato l'implementazione, utilizzando un timer fornito dalla libreria.

```
commit cc35bb1af8959f3dfb39b2dcc8c0a98021615001
Author: Mattia Papaccioli <mattiapapaccioli@gmail.com>
Date:   Fri Feb 6 12:04:58 2026 +0100

    fix: update labels with timer to avoid runtime error

diff --git a/src/main.c b/src/main.c
index 616e062..fcc48e4 100644
--- a/src/main.c
+++ b/src/main.c
@@ -154,6 +154,10 @@ const int padding_button = 50;
 const int height_button  = 50;
 const int width_button   = 50;

+static void update_current_temperature_cb(lv_timer_t* timer) {
+    update_current_temperature();
+}
+
 lv_obj_t * create_card(lv_obj_t * parent, const char * header_text,
lv_obj_t
 ** value_label_ptr, lv_color_t bg_color,
                       const lv_img_dsc_t * icon_dsc)
 {
@@ -341,11 +345,7 @@ int main(int argc, char ** argv)
     lv_img_set_src(decrement_icon, &minus);
     lv_obj_center(decrement_icon);

-    struct sigaction sa = {0};
-    sa.sa_handler = update_current_temperature;
-    sa.sa_flags   = SA_RESTART;
-    sigaction(SIGUSR1, &sa, NULL);
-    sigprocmask(SIG_UNBLOCK, &set, NULL);
+    lv_timer_create(update_current_temperature_cb, 1000, NULL);

     /* Enter the run loop of the selected backend */
     driver_backends_run_loop();
```

Codice 7 — `git show cc35bb1`

4.3.2 Troppi file descriptor aperti

Un altro problema l ho trovato invece nel modulo `pid-control`, specificatamente nella comunicazione con l'inverter tramite `modbus`. Ingenuamente, il codice effettuava una connessione a con `modbus` ogni volta che doveva settare la velocita della ventola dell inverter e dopo 1024 iterazioni circa, l' `ulimit` del sistema embedded, l'eseguibile crashava per i troppi file descriptor aperti. Per risolvere il problema e stato necessario semplicemente connettersi all'inverter una sola volta nel `main` del modulo `pid-control` invece che ripetutamente nella libreria che ho scritto per l "inverter.

figure(caption: `git show f0bbc5`, sourcecode[

```
commit e8f11fc2cf56682c62a2d3c52088442434524a92
Author: Mattia Papaccioli <mattiapapaccioli@gmail.com>
Date:   Fri Feb 6 17:10:30 2026 +0100

    fix: open file descriptor once

    bug fix that caused too many open files.
    the file was open every time we sent a command to the inverter.
    only one time is necessary.
    dont trust blindly llm code ;).

diff --git a/src/lib/devices/acs_310_modbus.c
b/src/lib/devices/acs_310_modbus.c
index 863b4de..869ff01 100644
--- a/src/lib/devices/acs_310_modbus.c
+++ b/src/lib/devices/acs_310_modbus.c
@@ -84,11 +84,7 @@ void check_faults(modbus_t *ctx) {
 bool send_command(modbus_t *ctx, int register_addr, uint16_t command) {
     if (!ctx) return false;

-    if (modbus_connect(ctx) == -1) {
-        LOG_ERROR("Connection failed: %s\n", modbus_strerror(errno));
-        modbus_free(ctx);
-        return false;
-    }
+

     int rc = modbus_write_register(ctx, register_addr, command);
     if (rc == -1) {
diff --git a/src/main.c b/src/main.c
index 08e2bf7..740dc78 100644
--- a/src/main.c
+++ b/src/main.c
@@ -14,6 +14,7 @@
 #include <sys/timerfd.h>
 #include <sched.h>
 #include <sys/stat.h>
+#include <errno.h>
```

```
 #include <common-control/common-control.h>
 #include <common-control/config.h>
@@ -225,6 +226,12 @@ int main()

     modbus_t *ctx = get_client();

+    if (modbus_connect(ctx) == -1) {
+        LOG_ERROR("Connection failed: %s\n", modbus_strerror(errno));
+        modbus_free(ctx);
+        return false;
+    }
+
     log_init();
     log_set_level(LOG_DEBUG);
     log_set_output(LOG_OUTPUT_CONSOLE);


])
```

5

Conclusione

5.1 Importanza del testing sul campo

I problemi che ho riscontrato durante la fase di installazione mi hanno mostrato il valore di eseguire test specifici sul campo con tutto il sistema collegato, a differenza della pratica dell'unit testing. Certi bug non si possono riprodurre con una singola parte del sistema ed e necessario aver in moto tutto per vedere dove si rompe.

5.2 Insegnamenti e considerazioni personali

Questo progetto mi ha insgnato molto. Anche se non centrale allo sviluppo del progetto ho imparato molto sul software `git`, che mi servira molto nel susseguirsi della mia carriera. Inoltre ho esplorato ed approfondito in dettaglio i microcontrollori, che personalmente mi piacciono molto. Ho imparato nozioni di ingegneria elettrica, che anche se non oggetto nel mio corso di studio si interlacciano con i microcontrollori. Ho provato con mano cosa significa applicare protocolli industriali come modbus, e vedere che con il mio computer posso far girare una ventola di grandi dimensioni mi ha gasato un botto.

A Riferimenti

[1] S. Raimbault, «libmodbus - A Modbus library for Linux, Mac OS X, FreeBSD, QNX and Windows». [Online]. Disponibile su: https://github.com/stephane/libmodbus

[2] L. Contributors, «LVGL». [Online]. Disponibile su: https://github.com/lvgl/lvgl

[3] M. Papaccioli, «Tesi». [Online]. Disponibile su: https://github.com/sbOogway/tesi

[4] M. Integrated, «DS18B20 Programmable Resolution 1-Wire Digital Thermometer». [Online]. Disponibile su: https://www.maximintegrated.com/en/products/sensors/DS18B20.html

[5] L. Contributors, «Linux template for LVGL». [Online]. Disponibile su: https://github.com/lvgl/lv_port_linux

[6] [Online]. Disponibile su: https://www.freedesktop.org/wiki/Software/libevdev/

[7] K. Inc., «CMake - Cross Platform Make». [Online]. Disponibile su: https://cmake.org/

[8] L. Torvalds e L. kernel developers, «The Linux Kernel Archives». [Online]. Disponibile su: https://www.kernel.org/

[9] B. contributors, [Online]. Disponibile su: https://buildroot.org/

[10] M. T. Inc., «AT91Bootstrap». [Online]. Disponibile su: https://github.com/linux4sam/at91bootstrap

[11] B. developers, «Barebox - bootloader for embedded systems». [Online]. Disponibile su: https://barebox.org/

[12] O. developers, «OpenSSH». [Online]. Disponibile su: https://www.openssh.com/

[13] N. T. Foundation, «Network Time Protocol - NTP». [Online]. Disponibile su: https://www.ntp.org/