StuFW Documentazione

Autore: Carlo Dormeletti

Versione di riferimento del firmware: **0.0.6**
Versione della guida: **0.01**
Data di stampa: **28 novembre 2020**

Indice

Prefazione

Questo documento è in fase sperimentale.

Al momento non vuole essere nulla in particolare, solo alcuni spunti per documentare il firmware.

Quindi prendetelo come qualcosa da leggere, senza nessuna garanzia di correttezza.

Principalmente serve per tenere traccia delle scoperte che si fanno sul firmware, mentre lo si modifica o lo si elabora.

Introduzione

Genesi del firmware

Questo firmware è nato per rendere semplice il firmware MK4duo, si è evoluto dopo la "morte" dello stesso e il suo passaggio nel limbo dei progetti abbandonati.

Nel compiere queste affermazioni, rendiamo il dovuto omaggio sia a MK4duo stesso, che al suo autore MagoKimbra Alberto Cotronei.

Nello spirito dell'antico motto: *"Il Re è morto. Viva il Re"* forkiamo e sviluppiamo secondo alcune linee guida, il paradigma che abbiamo in mente è quello di un firmware per studenti delle superiori per definizione squattrinati e con poche risorse, per cui:

- poche schede a 8 bit la RAMPS e la MKS Gen 1.2 - 1.4
- due display tra i più diffusi:
  - RepRapDiscount Full Graphics Controller (grafico a matrice di punti 128x64)
  - RepRapDiscount Smart Controller
- driver per stepper A4988 e DRV8825 diffusi ed economici
- poche funzioni con un occhio ai costi
- cercheremo di rendere semplici le operazioni di taratura e di messa in moto della stampante

La scelta di Mk4duo ci facilita le cose, in quanto è un ottimo firmware e permette di lavorarci sopra, ultima considerazione è stato forkato da Marlin per cui non compiamo nessun peccato di lesa maestà, continuiamo sulla strada dell'OpenSource.

Gli Autori

Capitolo 1

Analisi del firmware

Il firmware viene compilato usando queste direttive:

```
#include "StuFW.h"

void setup() {
  printer.setup();
}

void loop() {
  printer.loop();
}
```

potete notare come include un file **StuFw.h** che contiene gli header

All'interno di questo potete trovare una linea

```
#include "src/core/printer/printer.h"
```

che al suo interno definisce come "pubbliche" le due funzioni:

```
111   public: /** Public Function */
112
113     static void setup();  // Main setup
114     static void loop();   // Main loop
```

che poi vengono richiamate.

La parte interessante che poi costituisce il cuore del programma è la funzione **printer.loop()** che è definita nel file **printer.cpp**.

```
213 void Printer::loop() {
214
215   for (;;) {
216
217     printer.keepalive(NotBusy);
218
219     #if HAS_SD_SUPPORT
```

```
220
221       card.checkautostart();
222
223       if (card.isAbortSDprinting()) {
224         card.setAbortSDprinting(false);
225
226         #if HAS_SD_RESTART
227           // Save Job for restart
228           if (IS_SD_PRINTING()) restart.save_job(true);
229         #endif
230
231         // Stop SD printing
232         card.stopSDPrint();
233
234         // Clear all command in quee
235         commands.clear_queue();
236
237         // Stop printer job timer
238         print_job_counter.stop();
239
240         // Auto home
241         #if Z_HOME_DIR > 0
242           commands.enqueue_and_echo_P(PSTR("G28"));
243         #else
244           commands.enqueue_and_echo_P(PSTR("G28 X Y"));
245         #endif
246
247         // Disabled Heaters and Fan
248         thermalManager.disable_all_heaters();
249         zero_fan_speed();
250         setWaitForHeatUp(false);
251
252       }
253
254     #endif // HAS_SD_SUPPORT
255
256     commands.get_available();
257     commands.advance_queue();
258     endstops.report_state();
259     idle();
260
261   }
262 }
```


notate che alla fine richiama la funzione **idle()**, che è definita alla linea 440.

```
440 void Printer::idle(const bool ignore_stepper_queue/*=
                    ↪ false*/) {
441
442   lcdui.update();
```

In questo modo il firmware continua esegendo le funzioni.

Una piccola nota sui nomi:

ad esempio nella linea:

```
printer.keepalive(NotBusy);
```

viene chiamata la funzione **printer.keepalive** che però non troviamo in giro con quel nome.

Il nome più vicino è quello che troviamo in:

```
816 #if ENABLED(HOST_KEEPALIVE_FEATURE)
817
818   /**
819    * Output a "busy" message at regular intervals
820    * while the machine is not accepting
821    */
822   void Printer::keepalive(const BusyStateEnum state) {
823     if (!isSuspendAutoreport() && host_keepalive_watch.
                        ↪ stopwatch && host_keepalive_watch
                        ↪ .elapsed()) {
824       switch (state) {
```

Che però è nominata come **Printer::keepalive**.

Non è un mistero è semplicemente un artifizio che permette di nominare in modo diverso le cose, questo artifizio è fatto nel file **printer.h** con la linea

```
extern Printer printer;
```

piazzata proprio in fondo al file, questa permette di rinominare lo "spazio dei nomi" **Printer** come **printer** e quindi di utilizzare le funzioni chiamandole **printer.nomefunzione**.

I più scaltri di voi noteranno che però la funzione **keepalive** è si definita attorno alla linea 8228 che però è messa sotto la direttiva del preprocessore:

```
#if ENABLED(HOST_KEEPALIVE_FEATURE)
```

mentre viene richiamata senza particolari cautele all'inzio della funzione **loop**.

Il gioco viene fatto sempre nel file **printer.h**, attraverso le linee:

```
    #if ENABLED(HOST_KEEPALIVE_FEATURE)
      static void keepalive(const BusyStateEnum state);
    #else
```

```
      FORCE_INLINE static void keepalive(const
                          ↪ BusyStateEnum state) { UNUSED(
                          ↪ state); }
    #endif
```

che dichiarano la funzione **keepalive** ma usando l'else la dichiarano come FORCE_INLINE per cui questo riferimento è presente sia che sia che **HOST_KEEPALIVE_FEATURE** sia definito sia che non lo sia e il programma gira in modo corretto in entrambi i casi.

Un esempio ulteriore di questo meccanismo lo potere trovare in **src/lcd/lcdui.h**, dove vengono definite alcune cose sia per una stampante con LCD (grafico o a righe di caratteri) ma anche nel caso in cui non sia definito un LCD.

```
250     #else // NO LCD
251
252       static inline void init() {}
253       static inline void update() {}
254       static inline void set_alert_status_P(PGM_P message
                          ↪ ) { UNUSED(message); }
255       static inline void refresh() {}
256       static inline void set_status(const char* const
                          ↪ message, const bool persist=
                          ↪ false) { UNUSED(message);
                          ↪ UNUSED(persist); }
257       static inline void set_status_P(PGM_P const message
                          ↪ , const int8_t level=0) {
                          ↪ UNUSED(message); UNUSED(level);
                          ↪ }
258       static inline void status_printf_P(const uint8_t
                          ↪ level, PGM_P const fmt, ...) {
                          ↪ UNUSED(level); UNUSED(fmt); }
259       static inline void return_to_status() {}
260       static inline void reset_status() {}
261       static inline void reset_alert_level() {}
262       static constexpr bool has_status() { return false;
                          ↪ }
```

Ovviamente vengono definite come funzioni "vuote" oppure come funzioni che non compiono nulla, ma sono definite per poter essere richiamate liberamente senza ricorrere ad una marea di **#IF ENABLED** nel codice.