

Práctica 5

Objetivo:

Implementar un módulo de software para utilizar la UART y una MEF para parsear comandos recibidos por UART en **modo polling** (sin interrupciones ni DMA) usando HAL de STM32 (STM32F4 + STM32CubeIDE, C).

Punto 1

Implementar un módulo de software en un archivos fuente **API_uart.c** con su correspondiente archivo de cabecera **API_uart.h** y ubicarlos en el proyecto dentro de las carpetas /API/src y /API/inc, respectivamente.

En API_uart.h se deben ubicar los prototipos de las funciones públicas.

```
bool_t uartInit();  
void uartSendString(uint8_t * pstring);  
void uartSendStringSize(uint8_t * pstring, uint16_t size);  
void uartReceiveStringSize(uint8_t * pstring, uint16_t size);
```

En API_uart.c se deben ubicar los prototipos de las funciones privadas y la implementación de todas las funciones de módulo, privadas y públicas.

Consideraciones para la implementación:

1. uartInit() debe realizar toda la inicialización de la UART. Adicionalmente, debe imprimir por la terminal serie un mensaje con sus parámetros de configuración.

La función devuelve:

- True: si la inicialización es exitosa.
 - False: si la inicialización no es exitosa.
2. uartSendString(uint8_t *pstring) recibe un puntero a un string que se desea enviar por la UART completo (hasta el caracter '\0') y debe utilizar la función de la HAL HAL_UART_Transmit(...) para transmitir el string.
 3. uartSendStringSize(uint8_t * pstring, uint16_t size) recibe un puntero a un string que se desea enviar por la UART y un entero con la **cantidad de caracteres** que debe enviar. La función debe utilizar HAL_UART_Transmit(...) para transmitir el string.

Las funciones del módulo deben verificar TODOS los parámetros que reciben: para los punteros, se verifica que sean distintos a NULL y los parámetros de cantidad size deben estar acotados a valores razonables (¿cuáles?).

Se deben verificar los valores de retorno de TODAS las funciones del módulo UART de la HAL que utilicen.

Punto 2 (opcional)

Diseñar e implementar un módulo de software que implemente una MEF de parsing de comandos recibidos por UART. La MEF debe leer caracteres, formar líneas, reconocer comandos con argumentos, validar sintaxis y despachar acciones a funciones del sistema.

Se deben crear `API_cmdparser.c` y `API_cmdparser.h` en `/API/src` y `/API/inc`, respectivamente.

Se deben definir (dónde?)

```
#define CMD_MAX_LINE 64          // bytes, incluye terminador '\0'

#define CMD_MAX_TOKENS 5        // COMANDO + hasta 4 args

typedef enum {

    CMD_OK = 0,

    CMD_ERR_TIMEOUT,

    CMD_ERR_OVERFLOW,

    CMD_ERR_SYNTAX,

    CMD_ERR_UNKNOWN,

    CMD_ERR_ARG

} cmd_status_t;
```

Estados de la MEF

1. **CMD_IDLE**

Espera de caracteres. Transición a `CMD_RECV` al recibir cualquier carácter no-terminador.

2. **CMD_RECV**

Acumula en buffer hasta `\r` o `\n`.

- Si supera `CMD_MAX_LINE-1` → `CMD_ERROR(OVERFLOW)`.
- Si recibe terminador → `CMD_PARSE`.

3. **CMD_PARSE**

Tokeniza por espacios/tabs, normaliza mayúsculas del comando, valida número/forma de argumentos.

- Si válido → **CMD_EXEC**.
- Si inválido → **CMD_ERROR(SYNTAX|ARG|UNKNOWN)**.

4. **CMD_EXEC**

Invoca el *handler* correspondiente y retorna a **CMD_IDLE**.

5. **CMD_ERROR**

Emite mensaje de error descriptivo, limpia buffer y retorna a **CMD_IDLE**.

6. **CMD_TIMEOUT** (opcional pero recomendado)

Si no llegan terminadores en **T_LINE_TIMEOUT_MS** (p. ej. 1000 ms) estando en **CMD_RECV**, aborta la línea con **CMD_ERR_TIMEOUT**.

La MEF no debe bloquear; todo avance es por eventos (carácter recibido, timeout, fin de línea).

Reglas de protocolo

- Formato de línea: **COMANDO [arg1 [arg2...]]** finalizada en **\r\n**, **\n** o **\r**.
- **Case-insensitive** para el comando; tolera múltiples espacios.
- Líneas que comienzan con **#** o **//** → se ignoran.
- Respuestas del parser: cada mensaje termina con **\r\n**. Prompt opcional **>**.

Conjunto mínimo de comandos a reconocer → acciones encoladas

- **HELP** → **ACT_HELP**

- `LED ON|OFF|TOGGLE` → `ACT_LED` con `args.led.action` ∈ {ON, OFF, TOGGLE}
- `STATUS` → `ACT_STATUS`
- `BAUD?` → `ACT_BAUD_GET`
- `BAUD=<num>` (rango válido 9600..921600) → `ACT_BAUD_SET` con `args.baud_set.baud`
- `CLEAR` → `ACT_CLEAR`

Ante errores: `ERROR: unknown command`, `ERROR: bad args`, `ERROR: line too long`, `ERROR: timeout`.

Estados de la MEF (en `API_cmdparser.c`)

- `CMD_IDLE`: espera primer carácter no terminador → `CMD_RECV`.
- `CMD_RECV`: acumula hasta `\r/\n`.
 - Overflow → `CMD_ERROR` (imprime `ERROR: line too long`).
 - Terminador → `CMD_PARSE`.
 - Inactividad > `T_LINE_TIMEOUT_MS` → `CMD_TIMEOUT_STATE`.
- `CMD_PARSE`: tokeniza, normaliza comando, valida argumentos.
 - Válido → `CMD_EXEC`.
 - Inválido → `CMD_ERROR` (SYNTAX/ARG/UNKNOWN).
- `CMD_EXEC`: encola `cmd_action_t` correspondiente, imprime `OK\r\n` (o respuesta breve) y vuelve a `CMD_IDLE`.

- **CMD_ERROR**: imprime error, limpia buffer y vuelve a **CMD_IDLE**.
- **CMD_TIMEOUT_STATE**: imprime **ERROR: timeout**, limpia buffer y vuelve a **CMD_IDLE**.

No bloquear: `cmdPollUart()` procesa, por invocación, **hasta 16 bytes** (configurable) y retorna.