TP3

Features

- ***Display con FTM + DMA + Scatter Gather*:** Básicamente en el driver de la matriz de leds, al momento de actualizar la pantalla en vez de hacer una copia en memoria de toda la pantalla de 64x64, y hacer una transferencia DMA de todo eso directamente, se usa scatter and gather con dos buffers que se van alternando entre sí para escribir cada fila de la matriz.
  De esta forma, cuando se termina de escribir la n fila se genera una interrupción de major loop, se inicia la escritura automáticamente de la n+1 fila (por scatter and gather), y en la interrupción se llama a un callback registrado por el usuario de driver para actualizar el buffer de la n+2 fila.
  ¿Por qué? Del lado del driver ahorramos memoria, y con la dinámica de un callback para actualizar de una fila el usuario puede procesar ahí mismo la conversión entre display lógico y display en RGB. Nos apoyamos en el concepto ping pong buffer y la dinámica que usa **portaudio**.
  Ahorro en datos: Los bits de cada color, de cada led, son enviados mediante un ancho de pulso del FTM, configurado mediante el registro CnV del mismo (de 16 bits). Consecuentemente, de haber tenido la matriz completa guardada en memoria, se estaría usando:

$$2bytes \cdot 8\frac{bytes}{color} \cdot 3\frac{colores}{led} \cdot 64leds = 3kB$$

  Guardando sólo dos filas (16 leds), reducimos el uso de memoria 4 veces.
  Referencia: para referencia de esto, ver el driver *"MCAL/pwm_dma"* y luego el driver *"HAL/WS2812"*. Utilizan el driver de FTM "*MCAL/ftm*".
- ***Acelerómetro*:** Usamos la detección de orientación del FXOS8700CQ, que nos indica las siguientes orientaciones:
  - Portrait (up o down) o landscape (right o left)
  - Front o back

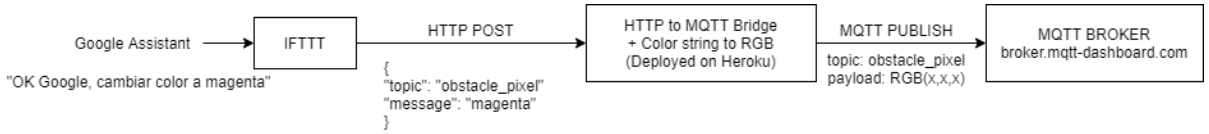
  Para eso tuvimos que configurar los registros (ver FXOS8700.c) PL_CFG, PL_COUNT, PL_BG_ZCOMP y PL_THS_REG y luego leemos la orientación en PL_STATUS. Además usamos una funcionalidad de antirebote y el bit "newlp" del PL_STATUS para solo detectar los cambios de orientación.
- ***Comunicación ESP8266 - K64F:*** La comunicación entre los microprocesadores se realiza vía UART, sobre lo cual en la capa de datos del modelo OSI se desarrolló un protocolo sencillo para garantizar sincronización independiente del momento de conexión, y una buena decodificación de los datos recibidos.

  START (stx) | TOPIC | DATA | END (eot)

  El protocolo cuenta con un byte de **start**, un byte de **end**, y luego además un byte de **escape** que permite evitar conflictos si el paquete utiliza los bytes de control como datos.
  El byte de **topic** permite decodificar qué tipo de mensaje se está mandando.
  Referencia, para referencia de esto, se desarrolló el protocolo en "lib/protocol" y se utilizó tanto en el ESP como el K64F, básicamente es una librería que provee el algoritmo de codificación y decodificación.
  ¿Lo podríamos mejorar? Sí, la realidad es que esto fue lo mínimo que necesitábamos para garantizar que no haya una mala decodificación de un paquete,

pero podría agregarse un campo de *checksum* que permita hacer una detección y/o corrección de errores, y además, quizá generar un *ack/nack* de respuesta para permitir que se detecte la ausencia de ambas partes en la comunicación punto a punto, por ejemplo en un timeout. O al mismo tiempo para permitir la retransmisión de paquetes que sean detectados como incorrectos.

- ***Comunicación con Google Home:*** Para poder conectar con Google home utilizamos la siguiente cadena de servicios:



  Luego el ESP y Node-Red se conectan al mismo broker y listo.
- ***Update del Display:*** Para hacer eficiente el refresco de la matriz, y consecuentemente disminuir la carga de la CPU, se decidió que la actualización del display fuera del tipo adaptativa. Es decir, se fija una tasa de refresco máxima de 60fps. En el caso de que en ese intervalo de tiempo no se hayan producido cambios en el board del juego, se ignora el evento de update display.