Protocolo de comunicación JPMSMRV maestro-esclavo de USB/UARTs de microcontroladores y Pcs 20200320A

En el diseño electrónico de diferentes placas va a existir un protocolo de comunicación entre microcontroladores o microcontroladores y PC.

Se va a crear (ya se está creando) una librería que permitirá comunicar de forma sencilla, eficiente y segura los diferentes dispositivos.

A esta librería la llamaré JPMSMRV, y esto es la versión 20200320A.

Esta librería consta de dos partes:
1.- COMUNICACIÓN ASCII
2.- MAPA DE REGISTROS VIRTUAL

1.- COMUNICACIÓN ASCII:

Realmente la parte de comunicación es una capa HAL que se utiliza de forma intermedia en las comunicaciones efectiva de los datos.

El sistema está formado por:
- Un maestro (microcontrolador o PC)
- Un esclavo (microcontrolador)

La comunicación se realizará con caracteres ASCII (los usados en la codificación base 64), con retornos de carro/avance de línea 0x0D 0x0A y CRC32 incluido en cada línea.

El protocolo no necesitará caracteres de inicio (start), ya que el primer carácter de la comunicación tras un inicio o tras un retorno de carro se interpretará como byte de inicio.
El CRC32 ya se encargará de descartar inicios inadecuados.

Al final este protocolo se basa en mandar y recibir líneas completas, que temporalmente se están almacenando en un búfer circular o un búfer de tamaño limitado de un tamaño adecuado.

Se define la trama de la línea de la siguiente manera:
B0 B1 B2 .. Bx 0x0D 0x0A

B0 a Bx contiene los caracteres ASCII de un bloque de datos binario codificado en "Base64", Bx.

El bloque binario está compuesto antes de ser codificado a base64 por un conjunto de bytes, considerándolos como "Datos" Dx:
D0 D1 .. Dx CRC0 CRC1 CRC2 CRC3

Como se ve se mantiene por convenio la transmisión de datos "Little-Endian", usado también en las arquitecturas Intel y ARM a la hora del almacenamiento en memoria o comunicaciones serie.

Para realizar el CRC32 se ha probado con éxito el uso de la librería Fast CRC32 de Stephan Brumme:
https://create.stephan-brumme.com/crc32/
https://create.stephan-brumme.com/crc32/Crc32.h
https://create.stephan-brumme.com/crc32/Crc32.cpp
Testeada tanto en un PC con procesador Intel/AMD como en el STM32F103C8T6 de la placa Bluepill.
https://create.stephan-brumme.com/crc32/Crc32Test.cpp

A la librería Fast CRC32 se le ha tenido que aplicar ciertas modificaciones para poder ser compilada correctamente.
En principio se usará el método "Slicing-by-8", por ser uno de los más rápidos y consumir no demasiados recursos en cuanto a memoria flash del STM32F103C8T6.
#define CRC32_USE_LOOKUP_TABLE_SLICING_BY_8
Usa +7328 bytes de flash respecto a CRC32_USE_LOOKUP_TABLE_BYTE en el STM32.
Usar slicing by 8 en el STM32F103C8T6 está consumiendo aproximadamente unos 8K de 64K, no es demasiado.

Con esta definición de trama ya se podría mandar un bloque de datos binarios de forma clara, trasparente y segura.
El receptor de los datos chequearía el CRC32 y admitiría el paquete de datos binarios o no.

En el caso de que no le llegue el paquete de datos o le llegue corrupto al receptor, se comportará como si no hubiera llegado.

El maestro se encargará de repetir las veces necesarias el paquete enviado y con los periodos de tiempo que se necesiten hasta que el esclavo devuelva una respuesta válida.
Más adelante se indicará qué es una respuesta válida.

Los datos binarios como hemos visto ya están compuestos por una serie de bytes, considerados "Datos" Dx:
D0 D1 D2 D3 D4 D5 .. Dx

Los primeros 4 bytes van a consistir en un identificador de paquete.
D0 D1 D2 D3 serian ID0 ID1 ID2 ID3

Lo que el paquete binario sería algo como esto:
ID0 ID1 ID2 ID3 D4 D5 .. Dx

Y al incluir el CRC32:
ID0 ID1 ID2 ID3 D4 D5 .. Dx CRC0 CRC1 CRC2 CRC3

Así el maestro numeraría todo los bloques binarios enviados, aun teniendo que recalcular el CRC32 de bloques binarios que se reenvían (por cambian el ID).

La respuesta del esclavo a todos los paquetes recibidos por el maestro va a costar de:
ID0 ID1 ID2 ID3 CRC0 CRC1 CRC2 CRC3
Todo esto codificado correctamente en base 64, y este paquete funciona de forma similar a un eco, omitiendo la retransmisión innecesaria de los datos.

Si el esclavo tuviera que mandar bloques de datos en respuesta a lo solicitado por el maestro los enviará posteriormente a este eco.
Estos bloques adicionales mantendrán el ID, los datos R referentes a la respuesta y el correspondiente CRC32.
ID0 ID1 ID2 ID3 R4 R5 .. Rx CRC0 CRC1 CRC2 CRC3

Estarse atentos, los datos sin el identificador, se han definido como:
- D3 D4 .. Dx
- R3 R4 .. Rx
Tanto R0 .. R3 como D0 .. D3 harán referencia a ID0 .. ID3.

El maestro tomará las decisiones adecuadas dependiendo de los paquetes recibidos, su integridad, el tiempo de recepción y los identificadores que pudieran estar desordenados.

Cualquier comunicación que no acabe con el eco correcto y la recepción de la respuesta adecuada se interpretará como un error en la comunicación y se repetirá, si se considerara procedente.

La librería que implementa esta capa HAL de momento se compone de funciones genéricas muy sencillas, no incluyendo las repeticiones o los tiempos.

En el envío, tanto el maestro como el esclavo usan la siguiente función:
void send_block (uint32_t id, uint8_t *bloque, uint32_t size);

En la recepción, tanto el maestro como el esclavo usarán:

uint32_t receive_block (uint32_t *id, uint8_t *bloque, uint32_t *size); // Siendo la respuesta distinta de 0x00 considerada como error.

En la recepción una respuesta 0x00 se puede considerar como que aun no ha habido error:
- Si size es 0x00 es que aun no se ha recibido nada o una línea completa, repitiéndose la comprobación.
- Si size es distinto distinto de 0x00 es que se ha recibido un bloque de datos de forma correcta.

Habiendo recibido un bloque de datos de forma correcta, en el caso de que el receptor sea el maestro, este bloque recibido puede ser un eco.
Hay que repetir el paso de recepción en el caso del maestro para recibir los datos del esclavo cuando así se requiera.

Hasta ahora la unidad básica de medida en la variable "size" es el "byte", esto es en esta parte de la comunicación.
Esta capa formada por estas 3 funciones ya contiene lo mínimo necesario para establecer una comunicación de forma correcta.

Posteriormente habrá que implementar más código para repeticiones, temporizaciones, descartar paquetes, etc.

Todo esto esta implementado en C, probado y compilado por el software Arduino para stm32duino y para PC con gcc.
Cuando estén más depuradas, testeadas las librerías y desarrolladas las aportaré.

2.- MAPA DE REGISTROS VIRTUAL:

Teniendo esta librería básica de comunicación, lo que se hace a continuación es crear un Mapa de Registros Virtual.

¿A qué me refiero con esto?

Imaginaros que hay que fijar 3 parámetros en el microcontrolador de la placa principal:
- Parámetro 1: Velocidad de flujo.
- Parámetro 2: Presión.
- Parámetro 3: Frecuencia respiratoria.

A la hora de implementar un programa principal o cuando el PC tenga que obtener los valores de estos parámetros sería bastante complicado estar revisando código y creando funciones específicas para la lectura de cada parámetro.
El Mapa de Registros Virtual simplifica mucho el desarrollo de código, la ampliación y el mantenimiento.

En el microcontrolador se define inicialmente la posición de memoria MRV_ADDR que va a tener cada parámetro, por ejemplo:
#define MRV_ADDR_VFLUJO 0x00000000
#define MRV_ADDR_PRESION 0x00000001
#define MRV_ADDR_FRESPIRATORIA 0x00000002
No es una dirección de memoria real del microcontrolador, es una posición de direccionamiento del Mapa de Registros Virtual.

Habrá una parte de código adicional que se encargara de asociar cada dirección de memoria del Mapa de Registros Virtual a la dirección de memoria real en el microcontrolador o PC, usando principalmente las direcciones de las variables o punteros.
#define MRV_MEM_ADRR_VFLUJO &velocidadflujo
#define MRV_MEM_ADRR_PRESION &presion
#define MRV_MEM_ADRR_FRESPIRATORIA &frecuenciarespiratoria

En el PC se mantienen estas mismas definiciones.
Cuando se solicite una lectura de un parámetro en el PC, por ejemplo el de presión, el PC estará haciendo:
- Lectura de la dirección 0x00000001 del Mapa de Registros Virtual.

Si se solicita una escritura en ese registro, se estaría cambiando el parámetro de presión.

Esto simplifica todo lo que se quiera implementar a:
1.- Definir (mapear) los parámetros, datos o bloques de datos que queramos usar, definiendo también el tamaño de cada bloque. Para 1 dato o parámetro el tamaño es 1 registro (ya indicaremos el tamaño de 1 registro).
2.- Una tabla "lookup table" o función que traslade las direcciones del MRV a las direcciones reales de memoria.
3.- Un comando de lectura de Mapa de Registros Virtual (MRV), indicando posición y tamaño.
4.- Un comando de escritura de Mapa de Registros Virtual (MRV), indicando posición y tamaño.

Ejemplo anterior completado:
#define MRV_ADDR_VFLUJO 0x00000000
#define MRV_SIZE_VFLUJO 0x00000001
#define MRV_MEM_ADRR_VFLUJO &velocidadflujo

```
#define MRV_ADDR_PRESION 0x00000001
#define MRV_SIZE_PRESION 0x00000001
#define MRV_MEM_ADRR_PRESION &presion
#define MRV_ADDR_FRESPIRATORIA 0x00000002
#define MRV_SIZE_FRESPIRATORIA 0x00000001
#define MRV_MEM_ADRR_FRESPIRATORIA &frecuenciarespiratoria
```

Como las arquitecturas usadas están optimizadas para 32 bits (o 64 bits) tanto en los ARM como el los procesadores de PC, todos los datos, registros y direcciones serán de 32 bits.
El usar como unidad en este Mapa de Registros Virtual un word de 32 bits como unidad mínima de almacenamiento tiene sus ventajas e inconvenientes.

Inconvenientes principales:
1.- Algo de procesamiento para bytes, cadenas char o de bytes, o words de 16 bits.
Si se quiere almacenar por ejemplo una matriz de 27 bytes en los registros se tendrán que utiliza 7 registros del MRV.
Cada registro contendrá 4 bytes, salvo el último, que contendrá únicamente 3 bytes.
2.- Se tiene que pensar que cada incremento de registro en bloques contiguos de datos supone un aumento de dirección de 4 bytes en el direccionamiento real de memoria del PC o microcontrolador.

Ventajas principales:
1.- En un sólo registro se puede almacenar datos de capturas de conversores de 16 bits o 24 bits de forma muy rápida, con pocas instrucciones de código ensamblador.
2.- Facilita el entendimiento del código, estando todo contenido en el MRV, independientemente de tratarse de tipos de datos distintos: byte, char, short int, int, floats, etc.
3.- Se puede usar variables "double" siempre y cuando se utilicen 2 registros por variable.

Aquí he priorizado lo práctico sobre la optimización de memoria.

Las 3 funciones del MRV son:
```
uint32_t mrv_read(uint32_t address, uint32_t *datos, uint32_t size);
uint32_t mrv_write(uint32_t address, uint32_t *datos, uint32_t size);
uint32_t mrv_cfg(char *port, uint32_t speed, uint32_t  databits, char *parity, uint32_t  stopbits);
```

Siendo la tercera la que se utiliza para seleccionar y configurar el canal de comunicaciones, tanto en el microcontrolador como en el PC.

Una respuesta distinta de 0x00 supondría un error, ya sea por errores en las comunicaciones o por cualquier otra causa.

Las 2 primeras funciones incluiría la capa HAL de comunicaciones vistas previamente.
Un PC podría ver el estado, modificar parámetros y datos o recibir un volcado de información de un microcontrolador de forma sencilla.
Todo basado en un Mapa de Registros Virtual.

He de indicar que el tamaño de memoria RAM disponible en el microcontrolador esta limitado a 20KB, así que hay que tener algo de cuidado con lo que se está utilizando en cada momento o lo que se reserva.
El búfer circular o un búfer de tamaño limitado de recepción de datos no puede ser demasiado grande, limitando la longitud máxima de cada línea a ese tamaño, y por tanto limitando el bloque máximo de datos trasmitidos por paquete a X bytes.

Teniendo bien definido el Mapa de Registros Virtual se puede desarrollar código de forma simultánea por diferentes programadores.

Todo se esta implementado en C, tanto para el software Arduino usado por stm32duino como para el gcc de los PC.
Cuando estén más depuradas, testeadas las librerías y desarrolladas las aportaré.

De momento, los desarrolladores que quieran empezar pueden usar esto, para poder compilar adecuadamente su código, aunque no se ejecute nada:

```
uint32_t mrv_read(uint32_t address, uint32_t *datos, uint32_t size) {
  return 0x00;
}
uint32_t mrv_write(uint32_t address, uint32_t *datos, uint32_t size) {
  return 0x00;
}
uint32_t mrv_cfg(char *port, uint32_t speed, uint32_t  databits, char *parity, uint32_t  stopbits) {
  return 0x00;
}
```

Respecto a la función de configuración mrv_cfg:
Por el tipo de dato no es posible fijar bits de stop a 1.5, esto raramente se utiliza.

Cuidado, el puerto seleccionado y la paridad son siempre cadenas de caracteres (strings), no variables numéricas.

Esta función tiene una forma de usarse diferente según en donde se utilice.

En windows se utilizará de forma similar a esta:

```
error=mrv_cfg("COM0",115200,8,"N",1);
```

En linux:

```
error=mrv_cfg("/dev/ttyACM0",115200,8,"N",1); // El driver USB del micro no usa /dev/ttyUSB0
```

En el microcontrolador STM32F103C8T6:

```
error=mrv_cfg("Serial",115200,8,"N",1);
error=mrv_cfg("Serial1",115200,8,"N",1);
error=mrv_cfg("Serial2",115200,8,"N",1);
error=mrv_cfg("USB",115200,8,"N",1); // Esto para acceder al mapa MRV del PC, auque esto no se utilizará, ya que el mapa MRV que se utiliza es el del micrcontrolador STM32.
```

Estas son las especificaciones iniciales de la librería JPMSMRV versión 20200320A, posiblemente sufra modificaciones sobre la marcha.