Librería y aplicación DLMS/COSEM para ESP32

Fecha	Versión	Descripción	Autor
7/11/2024	1.0.0	Primera versión del documento. - Versión librería DLMS/COSEM 1.0.0.LIB.ICESI - Versión aplicación de prueba de la librería 1.0.0.APP.ICESI	Cristian Rodriguez

Metrum Soluciones Inteligentes SAS
Calle 11 #100-121, Oficina 508, Edificio Campestre Towers
Tel: 602 405 64 37
e-mail: contacto@metrum.com.co
www.metrum.com.co
Cali - Valle

1. Introducción

Este Firmware consta de un *Driver o librería* en formato de archivo binario precompilado y una aplicación para su evaluación. El acceso a la librería está soportado sobre varias interfaces tipo header compatibles con la arquitectura ESP32:

1.1. Archivos de la librería

- libdlms.a: Archivo binario precompilado bajo la arquitectura ESP32 no compatible con otras variantes como ESP32-S, ESP32-C ni ESP32-H.
- conn.h: Archivo Header que incluye las funciones prototipo, estructuras y enumeraciones para interactuar con medidores DLMS/COSEM, específicamente el Iskra MT880.
- logServer.h: Archivo Header que incluye las funciones prototipo para la gestión del log (ver numeral 11).

1.2. Archivos de la aplicación de evaluación

- main.c: Archivo principal de la aplicación. Contiene la función main que inicializa el sistema de logging, conexión a un Access Point para actualización de la fecha y hora de la ESP32 y llamado a la función GatewayInit para crear el hilo encargado de leer/escribir el medidor.
- gatewayUart.c: Archivo con la lógica para acceder a la UART de la ESP32 e implementación del hilo que lee/escribe el medidor.
- gatewayUart.h: Archivo con la función prototipo GatewayInit.
- board.h: Archivo con las estructuras y la parametrización del Hardware que necesita el Driver para acceder al puerto serial de la ESP32 y que estará conectado al medidor.

2. Funciones soportadas

Todas las funcionalidades del Driver están expuestas dentro del archivo conn.h:

2.1. Identificadores, Firmware, medidas instantáneas y energías

- requestFirmwareAndSerial: Devuelve el serial y la versión de Firmware del medidor.
- requestInstantMeasures: Devuelve las medidas instantáneas, número de serie, fecha y hora, versión de Firmware y ángulos del medidor.
- requestEnergyMeasures: Devuelve las medidas de energías acumuladas, número de serie, fecha y hora y versión de Firmware del medidor.

2.2. Fecha y hora

- requestDateTime: Devuelve la fecha y hora del medidor.
- setDateTime: Ajuste de fecha y hora del medidor.

2.3. Perfiles de carga

- requestLoadProfile1: Devuelve el perfil de carga 1 del medidor.
- requestLoadProfile2: Devuelve el perfil de carga 2 del medidor.
- requestCapturePeriodLoadProfile1: Devuelve el periodo de captura en segundos del perfil de carga 1.
- requestCapturePeriodLoadProfile2: Devuelve el periodo de captura en segundos del perfil de carga 2.
- requestChannelsLoadProfile1: Devuelve los canales configurados del perfil de carga 1.
- requestChannelsLoadProfile2: Devuelve los canales configurados del perfil de carga 2.

2.4. Gestión de logs

Estas funciones tienen como propósito registrar la comunicación entre el Driver y el medidor de energía con enfoque de depuración o auditoría. Su uso es opcional.

- void* log_open: Inicializa los objetos necesarios para el registro de los Frames HDLC/DLMS-COSEM en un buffer externo.
- void log_close: Destruye los objetos asociados al log.

3. Conexión al medidor

Antes de realizar cualquier operación de lectura o escritura se debe indicar al Driver cómo comunicarse con el medidor. Para esto, primero se debe inicializar la estructura METER_CONNECTION con elementos como modelo del dispositivo, clave de acceso, tipo de seguridad, funciones *Callback* de acceso al puerto serial, etc. Las opciones por defecto para el medidor Iskra MT880 son:

```
METER_CONNECTION conn;
init_params(&conn);

/* Configure meter, passwords and callback */
conn.model = METER_MODEL_MT880;
conn.uartRead = &GatewayReadUart;
conn.uartWrite = &GatewayWriteUart;
conn.uartConfig = &GatewayConfigUart;
conn.retry = 3;
conn.addressSize = 2;
conn.serverPhyAddress = 31;
strcpy(conn.pass, "12345678"); /* Meter password *
```

Una vez que METER_CONNECTION está correctamente inicializado se pasa como argumento a las funciones detalladas en los numerales 2.1, 2.2 y 2.3. Su definición es:

- METER_MODEL model: Modelo del medidor. Se debe utilizar la enumeración METER_MODEL_MT880.
- char pass[65]: Clave o llave de acceso al medidor. Dependiendo de la operación a ejecutar, debe configurarse la clave/llave de lectura o escritura. Para medidores Iskra MT880 usar el valor “12345678”.
- char master[65]: Llave maestra para operaciones de actualización de llaves. No relevante para el medidor Iskra.
- char cipher[65]: Llave de cifrado en bloque para el acceso al medidor en modo mensajes cifrados y mensajes cifrados/autenticados. No relevante para el medidor Iskra.
- uint16_t serverPhyAddress: Dirección física del medidor. Si desea utilizar una dirección física diferente a la por defecto, que maneja internamente el Driver, indique el valor. Para medidores Iskra MT880 esta dirección se calcula sumando 16 a los dos últimos dígitos del número de serie.
- bool ignorePhyAddress: Ignorar la dirección física del medidor en la conexión HDLC. No relevante para el medidor Iskra.

- bool useServerLogAddress: Señala el uso de una dirección lógica personalizada del servidor. Si se activa, el valor para utilizar se indica en el parámetro serverLogAddress. No relevante para el medidor Iskra.
- uint16_t serverLogAddress: Dirección lógica del servidor personalizada. No relevante para el medidor Iskra.
- uint8_t addressSize: Utiliza un tamaño de dirección física HDLC personalizado para el servidor. Valores permitidos 0=automático, 1, 2 y 4 bytes. Para medidores Iskra MT880 usar el valor 2.
- serial_cll uartRead: Callback para acceder a la UART para operaciones de lectura.
- serial_cll uartWrite: Callback para acceder a la UART para operaciones de escritura.
- serial_config_cll uartConfig: Callback para reconfigurar la UART. No relevante para el medidor Iskra.
- void *log: Manejador del log. El Driver utiliza este manejador para copiar los Frames HDLC/DLMS-COSEM en un buffer externo (ver numeral 11).
- uint8_t retry: Configura el número de reintentos del último mensaje cuando no hay respuesta del medidor. El Driver calcula el tiempo de respuesta del medidor en milisegundos de la siguiente forma:

  ```
  Timeout = 25 * MIN_SERIAL_TIMEOUT
  ```

  Si pasado el tiempo anterior el Driver no recibe respuesta, envía un nuevo mensaje la cantidad de veces indicado en retry. La constante MIN_SERIAL_TIMEOUT se encuentra definida en el archivo gatewayUart.c, que se incluye como parte del ejemplo de aplicación, y se utiliza en la función GatewayConfigSerialPort al momento de configurar la UART de la ESP32.
- SECURITY_TYPE security: Tipo de seguridad usado en el intercambio de mensajes con el medidor. Por defecto, el Driver se comunica sin seguridad a menos que la aplicación se lo indique a través de este miembro. No relevante para el medidor Iskra.

Nota: Las operaciones de ajuste de fecha y hora requieren el uso de la clave de escritura del medidor. Las operaciones de lectura de identificadores, instantáneas, energías, ángulos y perfiles de carga requieren el uso de la clave de lectura del dispositivo. Por defecto la clave de escritura del medidor Iskra MT880 es "12345678".

Las opciones disponibles de modelos de medidor son:

- METER_MODEL_MT880: ISKRA MT880.

Las opciones disponibles del tipo de seguridad son (no relevante para el medidor Iskra MT880):

- SECURITY_TYPE_NONE: El intercambio de mensajes con el medidor será abierto, sin cifrado.
- SECURITY_TYPE_AUTHENTICATED_MESSAGES: El intercambio de mensajes con el medidor será abierto y su integridad será protegida mediante un tag de autenticación.
- SECURITY_TYPE_ENCRYPTED_MESSAGES: El intercambio de mensajes con el medidor será cifrado.
- SECURITY_TYPE_AUTHENTICATED_AND_ENCRYPTED_MESSAGES: El intercambio de mensajes con el medidor será cifrado y su integridad será protegida mediante un tag de autenticación.

Las funciones *Callback* uartRead, uartWrite y uartConfig son métodos de acceso al puerto serial para que el Driver pueda leer, escribir y reconfigurar la UART. Estas funciones deben tener la siguiente declaración.

```
typedef int (*serial_cll)(uint8_t *data, uint16_t size);
```

Observe el ejemplo de estas funciones *Callback*:

```
int GatewayWriteUart(uint8_t *pBuf, uint16_t size)
{
    int rc;
    rc = uart_write_bytes(uartHW, (char*) pBuf, size);
    uart_wait_tx_done(uartHW, portMAX_DELAY);
    return(rc);
}

int GatewayReadUart(uint8_t *pBuf, uint16_t size)
{
    int rc;
    rc = uart_read_bytes(uartHW, (uint8_t*) pBuf, size, waitTime);
    return rc;
}

static int GatewayConfigUart(uint8_t modeDLMS, uint16_t baudrate)
{
    int rc = 0;
    rc = GatewayConfigSerialPort(0, modeDLMS, baudrate);
    return rc;
}
```

El Driver a través de los argumentos *modeDLMS* y *baudrate* informan a la aplicación la manera en que debe reconfigurar, en tiempo de ejecución, la UART. Esta funcionalidad se utiliza para ciertos modelos de medidores que requieren alguna modificación en la UART al momento de accederlos. En la siguiente tabla se ilustra cómo debe parametrizarse la UART en función del modo. Puede encontrar un ejemplo de implementación en el archivo gatewayUart.c y debe utilizarse sin modificaciones para los medidores Iskra MT880.

modeDLMS	Parámetros UART
True	Velocidad = Valor que indica el Driver Bits de datos = 8 Paridad = Ninguna Bits de parada = 1
False	Velocidad = Valor que indica el Driver Bits de datos = 7 Paridad = Par Bits de parada = 1

4. Estructuras para almacenar datos del medidor

Para almacenar los datos obtenidos del medidor se emplea una estructura llamada METER_DATA. Los valores de los miembros que no estén soportados por el medidor el Driver les asigna el valor menos uno (-1).

Antes de utilizar la estructura METER_DATA, esta se debe inicializar con la función init_data:

```
METER_DATA data;
init_data(&data);
```

METER_DATA se incluye, junto a METER_CONNECTION, como argumento en las funciones de lectura de datos (ver secciones 6 y 7). Después de que las funciones de lectura retornan de su llamado, se pueden acceder a los datos leídos mediante los siguientes miembros:

- uint32_t datetime: Fecha y hora del medidor.
- char serial[64]: Número de serie del medidor en formato ASCII.
- char firmware[64]: Versión de Firmware en formato ASCII.
- uint32_t *dateTimeOfInterval: Arreglo con la fecha y hora del intervalo.
- float *valueOfInterval: Arreglo con los valores de los intervalos.
- uint16_t totalIntervals: Número de intervalos obtenidos.
- char *obisChannels: Arreglo con la lista de códigos OBIS de los canales del perfil de carga.
- uint8_t totalChannels: Número de canales obtenidos, tanto para la lectura del perfil de carga como para la lectura de los canales.
- float voltage[3]: Voltaje en las 3 fases, siendo [0] fase A, [1] fase B y [2] fase C. En medidores monofásicos solo el valor de la fase 1 es válido.
- float current[3]: Corriente en las 3 fases, siendo [0] fase A, [1] fase B y [2] fase C. En medidores monofásicos solo el valor de la fase 1 es válido.
- float frequency: Frecuencia principal.
- float power_factor[3]: Factor de potencia en las 3 fases, siendo [0] fase A, [1] fase B y [2] fase C. En medidores monofásicos solo el valor de la fase 1 es válido.
- float active_power_total_import: Potencia activa total importada.
- float active_power_total_export: Potencia activa total exportada.
- float reactive_power_total_import: Potencia reactiva total importada.
- float reactive_power_total_export: Potencia reactiva total exportada.
- float apparent_power_total: Potencia aparente total.
- float active_power_import[3]: Potencia activa importada en las 3 fases, siendo [0] fase A, [1] fase B y [2] fase C. En medidores monofásicos solo el valor de la fase 1 es válido.

- float active_power_export[3]: Potencia activa exportada en las 3 fases, siendo [0] fase A, [1] fase B y [2] fase C. En medidores monofásicos solo el valor de la fase 1 es válido.
- float reactive_power_import[3]: Potencia reactiva importada en las 3 fases, siendo [0] fase A, [1] fase B y [2] fase C. En medidores monofásicos solo el valor de la fase 1 es válido.
- float reactive_power_export[3]: Potencia reactiva exportada en las 3 fases, siendo [0] fase A, [1] fase B y [2] fase C. En medidores monofásicos solo el valor de la fase 1 es válido.
- float reactive_power_total_qi: Potencia reactiva total cuadrante I.
- float reactive_power_total_qii: Potencia reactiva total cuadrante II.
- float reactive_power_total_qiii: Potencia reactiva total cuadrante III.
- float reactive_power_total_qiv: Potencia reactiva total cuadrante IV.
- float active_energy_total_import: Energía activa total importada.
- float active_energy_total_export: Energía activa total exportada.
- float reactive_energy_total_import: Energía reactiva total importada.
- float reactive_energy_total_export: Energía reactiva total exportada.
- int capturePeriod: Periodo de captura del perfil de carga.
- float voltageAngleB2A: Ángulo de voltaje entre Fase B y A.
- float voltageAngleC2A: Ángulo de voltaje entre Fase C y A.

5. Códigos de error

Todas las funciones del Driver devuelven un código de error para conocer el resultado de la operación. Estos códigos se agrupan en la enumeración DRIVER_ERROR_CODES y se describen a continuación:

- DRIVER_ERROR_OK = 0: La operación se ejecutó sin errores.
- DRIVER_ERROR_PASSWORD = -1: La clave de acceso no es el correcto.
- DRIVER_ERROR_METER_NOT_RESPOND = -2: El medidor no responde.
- DRIVER_ERROR_READING_OBJECT = -3: Error al leer el objeto.
- DRIVER_ERROR_DATA_INVALID = -4: La función recibió un argumento no válido.
- DRIVER_ERROR_METHOD = -5: Error al ejecutar la desconexión/conexión del suministro.
- DRIVER_ERROR_WRITE = -6: Error al escribir el objeto.
- DRIVER_ERROR_NOT_SUPPORTED = -7: Función no soportada por el medidor.
- DRIVER_ERROR_OUT_OF_MEMORY = -8: No hay memoria para guardar los eventos.
- DRIVER_ERROR_NOT_IMPLEMENTED = -9: Modelo de medidor no soportado.

6. Lectura de registros y parámetros

Las siguientes funciones se emplean para obtener las medidas instantáneas, el serial, la fecha y hora, la versión de Firmware, los registros de energías acumulados, el periodo de captura de los perfiles de carga 1 y 2 y el ángulo entre voltajes:

- requestFirmwareAndSerial
- requestInstantMeasures
- requestEnergyMeasures
- requestDateTime,
- requestCapturePeriodLoadProfile1
- requestCapturePeriodLoadProfile2

```
if (requestFirmwareAndSerial(&conn, &data) != 0) {
    ESP_LOGE(TAG_GW, "requestFirmwareAndSerial error");
    continue;
}
```

```
if (requestInstantMeasures(&conn, &data) != 0) {
    ESP_LOGE(TAG_GW, "requestInstantMeasures error");
    continue;
}
```

```
if (requestEnergyMeasures(&conn, &data) != 0) {
    ESP_LOGE(TAG_GW, "requestEnergyMeasures error");
    continue;
}
```

```
if ((err = requestDateTime(&conn, &data)) != 0) {
     ESP_LOGE(TAG_GW, "requestDateTime error %d", err);
}
```

```
if ((err = requestCapturePeriodLoadProfile1(&conn, &data)) != 0) {
     ESP_LOGE(TAG_GW, "requestLoadProfileCapturePeriod 1 error %d", err);
}
```

7. Lectura del perfil de carga

Las funciones de lectura del perfil de carga tienen como parámetros el tiempo de fin del intervalo, en formato UNIX, y el número de horas previos a filtrar:

```
uint32_t dateTimeEnd = 1616976000;


if ((err = requestLoadProfile1(&conn, &data, dateTimeEnd, 1)) != 0) {
   ESP_LOGE(TAG_GW, "requestLoadProfile1 error %d", err);
}
```

```
uint32_t dateTimeEnd = 1616976000;


if ((err = requestLoadProfile2(&conn, &data, dateTimeEnd, 1)) != 0) {
   ESP_LOGE(TAG_GW, "requestLoadProfile2 error %d", err);
}
```

El número de horas a consultar dependerá de la cantidad de memoria disponible en el sistema embebido. Hasta 4 horas se puede consultar sin que el sistema se quede sin recursos.

La información de los intervalos consultados al medidor se accede de la siguiente forma:

- El miembro dateTimeOfInterval, de la estructura METER_DATA, devuelve la fecha y hora del intervalo bajo el formato de tiempo Unix.
- La cantidad de intervalos recibidos se consultan en el miembro totalIntervals de METER_DATA.
- La cantidad de canales capturados se consultan en el miembro totalChannels de METER_DATA. El número máximo de canales a consultar podrá ser modificado mediante la definición MAX_CHANNELS que está ubicada en el archivo conn.h. Por defecto, el valor está limitado a 32. Cada canal estará identificado por un código OBIS en formato String de máximo 24 caracteres terminados en NULL.
- El miembro valueOfInterval, de la estructura METER_DATA, devuelve los valores de los intervalos en formato flotante con escala aplicada.

```
dateTimeEnd = 1628604900;
init_data(&data);
/* Get load profile 1 */
if ((err = requestLoadProfile1(&conn, &data, dateTimeEnd, 4)) != 0) {
   ESP_LOGE(TAG_GW, "requestLoadProfile1 error %d", err);
}
```

```
for (i = 0; i < data.totalIntervals; i++)
{
   timer = data.dateTimeOfInterval[i];
   tm_info = localtime(&timer);
   memset(buffer, 0, sizeof(buffer));
   strftime(buffer, 26, "%Y-%m-%d %H:%M:%S", tm_info);
   ESP_LOGI(TAG_GW, "Date and time of interval %s", buffer);

   for (c = 0; c < data.totalChannels; c++)
   {
      ESP_LOGI(TAG_GW, "OBIS %s\tValue\t%8.2f", data.obisChannels + (c * OBIS_LENGTH),
(data.valueOfInterval + c)[i * MAX_CHANNELS]);
   }
}
free(data.dateTimeOfInterval);
free(data.valueOfInterval);
free(data.obisChannels);
```

Nota: Después de finalizada la lectura del perfil de carga, se debe liberar los datos de la memoria del sistema ejecutando un free() sobre los puntero dateTimeOfInterval, valueOfInterval y obisChannels.

En memoria, la fecha y hora del intervalo se organiza como dateTimeOfInterva[numero_intervalo]:

- dateTimeOfInterva[0] = fecha y hora del intervalo 0
- dateTimeOfInterva[2] = fecha y hora del intervalo 2
- dateTimeOfInterva[4] = fecha y hora del intervalo 4

En memoria, el valor del intervalo se organiza como (valueOfInterval + numero_canal)[numero_intervalo]:

- (valueOfInterval + 0)[4] = Valor del canal 0 del intervalo 4
- (valueOfInterval + 1)[0] = Valor del canal 1 del intervalo 0

El código OBIS del canal se consulta en el arreglo obisChannel. En memoria, el código se organiza como (obisChannels + (numero_canal + OBIS_LENGTH)). OBIS_LENGTH es una constante cuyo valor siempre es 25:

- obisChannels + (0 * OBIS_LENGTH) = Código OBIS del canal 0.
- obisChannels + (11 * OBIS_LENGTH) = Código OBIS del canal 11.

Si la aplicación requiere aumentar la cantidad de canales a leer, edite la definición MAX_CHANNELS del archivo conn.h. El aumento en el número de canales conlleva a mayor uso de memoria RAM.

```
#define MAX_CHANNELS  32// Adjust to set the maximum number of channels
including time
```

8. Lectura de los canales del perfil de carga

Las funciones de lectura de los canales del perfil de carga 1 y 2 son simples y pueden ejecutarse como se indica a continuación.

```
if ((err = requestChannelsLoadProfile1(&conn, &data)) != 0) {
   ESP_LOGE(TAG_GW, "requestChannelsLoadProfile1 error %d", err);
}
```

```
if ((err = requestChannelsLoadProfile2(&conn, &data)) != 0) {
   ESP_LOGE(TAG_GW, "requestChannelsLoadProfile2 error %d", err);
}
```

La información de los canales consultados al medidor se accede de la siguiente forma:

- El miembro channels, de la estructura METER_DATA, devuelve la descripción de los canales que incluye:
  - Código OBIS del objeto o canal
  - Clase del objeto o canal
    - Dato
    - Registro
    - Registro extendido
    - Demanda
  - Tipo de atributo del objeto o canal
    - Ninguno
    - Valor
    - Promedio actual (solo para objetos tipo demanda)
    - Último promedio (solo para objetos tipo demanda)
  - Escala
  - Unidad
- La cantidad de canales capturados se consultan en el miembro totalChannels de METER_DATA.

El miembro channels es un puntero a una estructura de tipo CHANNEL_DESCRIPTION cuya definición es la siguiente:

```
typedef struct {
   char obis[OBIS_LENGTH];
   OBJECT_CLASS_TYPE class;
   ATTRIBUTE_INDEX_TYPE attribute;
   int scale;
   UNIT_TYPE unit;
```

```
} CHANNEL_DESCRIPTION;
```

- El formato del código OBIS es un String de máximo 24 caracteres terminados en NULL.
- El tipo o clase del objeto o canal es una enumeración de tipo OBJECT_CLASS_TYPE.
- El tipo de atributo capturado es una enumeración de tipo ATTRIBUTE_INDEX_TYPE.
- La escala del objeto es un entero identificado por el miembro scale. El cálculo del multiplicador de escala se obtiene mediante la siguiente fórmula:
    - 10^(scale)
- El tipo de unidad es una enumeración de tipo UNIT_TYPE.

Los posibles valores de la clase del objeto o canal son:

```
typedef enum {
    OBJECT_CLASS_TYPE_NONE = 0,
    OBJECT_CLASS_TYPE_DATA,
    OBJECT_CLASS_TYPE_REGISTER,
    OBJECT_CLASS_TYPE_EXTENDED_REGISTER,
    OBJECT_CLASS_TYPE_DEMAND_REGISTER,
} OBJECT_CLASS_TYPE;
```

Los posibles valores del tipo de atributo del canal son:

```
typedef enum {
    ATTRIBUTE_INDEX_NAME_NONE = 0,
    ATTRIBUTE_INDEX_NAME_VALUE,
    ATTRIBUTE_INDEX_NAME_CURRENT,
    ATTRIBUTE_INDEX_NAME_LAST,
} ATTRIBUTE_INDEX_TYPE;
```

Algunos posibles valores del tipo de unidad son (la lista completa se encuentra en conn.h):

```
typedef enum {
    UNIT_TYPE_YEAR = 1,
    UNIT_TYPE_MONTH = 2,
    UNIT_TYPE_WEEK = 3,
    UNIT_TYPE_DAY = 4,
    UNIT_TYPE_HOUR = 5,
    ...
    UNIT_TYPE_VAR = 29,
```

```
    UNIT_TYPE_WATT_HOUR = 30,
    UNIT_TYPE_VOLT_AMPERE_HOUR = 31,
    UNIT_TYPE_VAR_HOUR = 32,
    ...
  } UNIT_TYPE;
```

Para acceder correctamente a la descripción de los canales puede llamar a cada objeto como si fuera un Array:

```
for (c = 0; c < data.totalChannels; c++)
{
   ESP_LOGI(TAG_GW, "Capture object [%d] OBIS %s Class %d, Index %d Scale %d Unit %d",
c, data.channels[c].obis, data.channels[c].class, data.channels[c].attribute,
data.channels[c].scale, data.channels[c].unit);

}

free(data.channels);
```

Nota: Después de finalizada la lectura de los canales del perfil de carga, estos se deben liberar de la memoria del sistema ejecutando un free() sobre el puntero channels.

9. Ajuste de fecha y hora

La función de ajuste de fecha y hora recibe como parámetro especial la nueva hora en formato Unix. Su ejecución requiere que la estructura METER_CONNECTION tenga asignado el password de escritura.

```
/* Set datetime: Sunday, March 28, 2021 1:34:36 PM GMT-05:00 */
if (setDateTime(&conn, &data, 1616956476) != 0) {
    ESP_LOGE(TAG_GW, "setDateTime error");
}
```

10. Lectura del periodo de captura del perfil de carga

Las funciones de lectura del periodo de captura del perfil de carga devuelven el valor en segundos. Su ejecución requiere que la estructura METER_CONNECTION tenga asignado el password de escritura.

```
if ((err = requestCapturePeriodLoadProfile1(&conn, &data)) != 0) {
  ESP_LOGE(TAG_GW, "requestLoadProfileCapturePeriod 1 error %d", err);
}
```

```
if ((err = requestCapturePeriodLoadProfile2(&conn, &data)) != 0) {
  ESP_LOGE(TAG_GW, "requestLoadProfileCapturePeriod 2 error %d", err);
}
```

11. Log de Frames HDLC/DLMS-COSEM

El Driver provee la funcionalidad de respaldar los Frames HDLC/DLMS-COSEM directamente en un buffer externo. Para esto, la aplicación debe suministrar los recursos de memoria necesarios para que el Driver entregue los mensajes a la tarea que los requiera.

11.1. Creación del log

La creación del log se realiza con la función log_open. Como argumento recibe los siguientes parámetros:

- TaskHandle_t handle: Manejador de la tarea externa que recibirá los mensajes.
- SemaphoreHandle_t *mutex: Puntero a un semáforo para sincronizar el acceso al buffer compartido entre el Driver y la tarea externa que recibirá los mensajes. La aplicación debe inicializar este semáforo antes de ejecutar la función **log_open.**
- const int maxSizeBuffer: Tamaño máximo del buffer donde se copiarán los mensajes. El Driver utiliza este parámetro para no rebasar los límites de memoria del buffer compartido.
- char buffer[maxSizeBuffer]: Buffer compartido donde se copiarán los mensajes.
- int *sizeReceived: Puntero a una variable compartida para indicar el número de bytes recibidos. El Driver utiliza este puntero para comunicar a la tarea externa el número de bytes que se han escrito en el buffer compartido. Es responsabilidad de la tarea externa limpiar este valor una vez el mensaje ha sido procesado por la aplicación.

Ejemplo de creación:

```
#define MAX_SIZE_FRAME  1024
char frameHDLC[MAX_SIZE_FRAME];
int frameSize;
TaskHandle_t printTaskHd;
SemaphoreHandle_t xMutex = NULL;

void GatewayUartFxn(void *arg0)
{
   xMutex = xSemaphoreCreateMutex();
   xTaskCreate(print_frames_task, "print_frames_task", 4096, NULL, 1, &printTaskHd);

   void* log = log_open(printTaskHd, &xMutex, frameHDLC, &frameSize, MAX_SIZE_FRAME);
  if (log == NULL)
  {
     ESP_LOGE(TAG_GW, "Log creation error");
  }
  ...
}
```

El resultado de la creación del log es un manejador de tipo void*. Para validar si la creación fue exitosa la aplicación debe verificar que el valor del puntero sea diferente de nulo:

El parámetro maxSizeBuffer puede modificarse para encontrar el máximo valor que le permita al Driver copiar un mensaje completo incluyendo las estampas de tiempo.

11.2. Destrucción del log

La destrucción del log tiene como fin liberar recursos en el sistema y se realiza con la función **log_close**. Como parámetro recibe el manejador obtenido en la función log_open.

```
log_close(log);
```

Nota: La destrucción del log no libera los recursos relacionados con los parámetros que recibe la función log_open, como lo son el semáforo, el buffer y la tarea.

11.3. Inicialización del Driver para uso del log

Para que el Driver pueda utilizar el log, se debe inicializar el miembro log de la estructura METER_CONNECTION con el manejador obtenido de la función log_open. El Driver automáticamente registrará y estampará cualquier mensaje que transmita o reciba del medidor y lo copiará en el buffer compartido que se pasó como parámetro en la función log_open.

```
METER_CONNECTION        conn;
init_params(&conn);
conn.log = log_open(printTaskHd, &xMutex, frameHDLC, &frameSize, MAX_SIZE_FRAME);
```

11.4. Recepción de los mensajes

La recepción de los mensajes es sencilla y se realiza desde la tarea cuyo manejador se indicó como parámetro en la función log_open. Cuando el Driver tiene listo un mensaje recibido o transmitido al medidor notificará a la tarea externa mediante un evento del sistema. La tarea externa esperará la recepción de este evento antes de entrar a consultar el buffer compartido.

En el siguiente se ejemplo se ilustra la recepción de los mensajes del Driver:

```
void print_frames_task(void *arg0)
{
    BaseType_t xResult;
    uint32_t ulNotifiedValue;
    while (true)
    {
```

```
        xResult = xTaskNotifyWait( pdFALSE,
                ULONG_MAX,
                &ulNotifiedValue,
                portMAX_DELAY);
        if(xResult == pdPASS)
        {
           xSemaphoreTake(xMutex, portMAX_DELAY);
           printf(frameHDLC);
           memset(frameHDLC, 0, MAX_SIZE_FRAME);
           frameSize = 0;
           xSemaphoreGive(xMutex);
        }
     }
}
```

Nota: Observe que en el ejemplo anterior la tarea externa bloquea indefinidamente su ejecución hasta recibir la notificación del Driver. Sin embargo, el usuario puede agregar un tiempo de espera modificando el parámetro portMAX_DELAY por otro valor que se ajuste a la necesidad de la aplicación.

Debido a que el buffer compartido y la variable sizeReceived son recursos gestionados por el Driver y la tarea externa, es necesario protegerlos para evitar comportamientos indefinidos en la aplicación por su acceso concurrente. Para esto se emplea el semáforo que se pasó como parámetro en la función log_open:

```
xSemaphoreTake(xMutex, portMAX_DELAY);
printf(frameHDLC);
memset(frameHDLC, 0, MAX_SIZE_FRAME);
frameSize = 0;
xSemaphoreGive(xMutex);
```

Nota: Tenga presente que es responsabilidad de la tarea externa limpiar la variable sizeReceived una vez el mensaje ha sido procesado por la aplicación. Si no se limpia esta variable, el Driver dejará de copiar nuevos mensajes en el buffer compartido.

11.5. Formato de mensajes

Los mensajes se encuentran estampados en formato Unix y etiquetados de la siguiente forma:

- TX: Frame transmitido al medidor.
- RX: Frame recibido del medidor.

Cada línea se encuentra delimitada por los caracteres “\r\n”.