Comentario técnico: CTC-098
Componente: **archivos de log de Mongoose-OS con ESP32**
Autor: Sergio R. Caprile, Senior R&D Engineer

| Revisiones | Fecha | Comentarios |
|---|---|---|
| 0 | 09/06/20 | |
| | | |
| | | |

Mongoose-OS incorpora un esquema de log en archivos rotativos que además provee un acceso mediante la ya varias veces mencionada API del tipo RPC (Remote Procedural Call), a la cual podemos acceder y utilizar a través de diversos transportes. Este esquema de bitácora puede tanto tomar mensajes del sistema como los que nuestra aplicación voluntariamente genera, lo cual nos resulta una excelente herramienta “forense” a la hora de intentar inferir por qué pasó lo que no debía haber pasado.

## File-logger

Se trata de un esquema similar al encontrado tradicionalmente en los servidores, en el que un grupo de archivos se va llenando de a uno por vez con mensajes provenientes de diversas fuentes (y en respuesta a sucesos que ameritan la escritura), cambiando de archivo cuando se llega a un determinado tamaño. Cuando se alcanza una cierta cantidad de archivos, el esquema borra el último. Esto funciona en cierta forma como un buffer circular pero sin la complejidad asociada.

## Configuración

Configuramos el ESP32 con Mongoose-OS para operar como un Access Point con un web server. La operación como AP nos resulta la forma tal vez más rápida y simple de realizar las pruebas y explicar la operación. La configuración puede realizarse manualmente mediante RPC, en un archivo de configuración JSON; o definirla en el archivo YAML que describe el proyecto. Para las pruebas elegimos esta última opción.

```
libs:
  - origin: https://github.com/mongoose-os-libs/file-logger  # Incluye el file-logger
config_schema:
  - ["file_logger.enable", true]         # Habilita la operación del file-logger
  - ["file_logger.prefix", "log_"]       # Prefijo para los nombres de los archivos
  - ["file_logger.max_file_size", 10000] # Tamaño máximo de cada archivo
  - ["file_logger.max_num_files", 5]     # Cantidad máxima de archivos (<=100)
  - ["file_logger.timestamps", false]    # Veremos timestamps más adelante
  - ["file_logger.level", 2]             # Nivel de log
  - ["file_logger.include", "adentro,tambien"] # Lista de substrings para filtrar mensajes
  - ["file_logger.syslog_enable", true]  # Captura los mensajes del sistema
  - ["file_logger.rpc_service_enable", true] # Habilita RPC adicionales para manejo y acceso
```

### Prefijo

Los nombres de archivo comienzan con este prefijo, y continúan con la fecha de creación.

### Nivel de log

El sistema tiene cuatro niveles de log y un esquema de prioridades. Utilizando las funciones de log podemos loggear un determinado texto indicando su correspondiente nivel; el sistema redirige este texto a los logs cuando su nivel de prioridad lo permite (el número es igual o menor). Luego de este filtro, el texto se vería por

ejemplo por la UART y es a continuación filtrado una vez más; si no tiene la prioridad permitida no pasa a los archivos.
Uno de los parámetros del sistema es `debug.level`; los niveles posibles son, en orden de prioridad decreciente:

- `ERROR (0)`
- `WARN  (1)`
- `INFO  (2)`
- `DEBUG (3)`
- `VERBOSE_DEBUG (4)`

Así, el nivel 2 configurado (que coincide con el que por defecto utiliza el sistema en `debug.level`) permite mensajes `INFO`, `WARN` y `ERROR`.
En nuestro código generaremos los registros llamando a las funciones:

- en mJS: `Log.print(Log.`*`level`*`, texto)`
  - por ejemplo: `Log.print(Log.WARN, 'Falla el ' + algo);`
- en C: `LOG(LL_`*`level`*`, (textosprintf))` (en realidad es una macro)
  - por ejemplo: `LOG(LL_WARN, ("Falla el %s", algo));`

### Substrings

Es posible aplicar además un filtro adicional basado en que el texto debe contener al menos una de las cadenas que figuran en este parámetro. Con la configuración del ejemplo sólo se almacenan los textos que contienen 'adentro' o 'tambien'[1]. Podemos de esta forma limitar el uso sólo a determinados módulos utilizando su nombre. Si no deseamos usarlo, podemos omitir este parámetro o poner una cadena vacía.

### Mensajes del sistema

Si lo habilitamos, los mensajes generados por Mongoose-OS pasan a los archivos de log, pero siguen el mismo esquema de filtrado, es decir, deberán tener prioridad suficiente y contener alguno de los substrings configurados.

## Operación

Luego de compilado el código (`mos build`) y grabado el microcontrolador (`mos flash`) mediante *mos tool*, observaremos en el log la dirección del Access Point y el nombre.

```
[Jun  2 14:39:50.935] esp32_wifi.c:450        WiFi AP: SSID myESP_807A98, channel 6
[Jun  2 14:39:51.802] esp32_wifi.c:507        WiFi AP IP: 192.168.4.1/255.255.255.0 gw [...]
[Jun  2 14:39:51.813] mgos_http_server.c:343  HTTP server started on [80]
[Jun  2 14:39:51.880] init.js:6               ### init script started ###
```

A continuación, nos conectaremos a esa red.
En los archivos provistos hemos incluido algunas RPC que nos permitirán registrar eventos; en las mismas se registran textos con y sin las cadenas configuradas, de modo de poder observar el funcionamiento. Dichas RPC llevan los nombres alegóricos `Do.Adentro`, `Do.Tambien`, y `Do.Afuera`, permitiendo inferir con poco margen de error cuál genera un texto que queda adentro del log, cuál también, y cuál genera un texto que queda afuera del registro.
Luego de generados algunos eventos, podemos ingresar al módulo, observar la cantidad de archivos disponibles, cuál es el más antiguo y cuál el más reciente, llamando a la RPC `FileLog.Status` :

```
$ mos --port http://192.168.4.1/rpc call FileLog.Status
{
```

1 Evitamos posibles problemas de localización sacrificando acentos y manteniéndonos en el ASCII tradicional.

```
  "enable": true,
  "num_files": 1,
  "oldest": "//log_000-19700101-000001.log",
  "newest": "//log_000-19700101-000001.log"
}
```

Por supuesto que también podemos llamar a la RPC `File.List` y observar el listado de todos los archivos.
Según como hayamos configurado nuestro esquema (no en este ejemplo), el acceso para traer logs puede a su vez generar logs. También, en condiciones reales, nuestro acceso puede ser concurrente con un proceso de escritura. Por estas razones, el proveedor recomienda el siguiente procedimiento:

- Detener el logging configurando `file_logger.enable=false`.
  - por ejemplo: `mos call Config.Set '{"config": {"file_logger": {"enable": false } }, "reboot": false, "save": false}'` realiza esta operación y no graba la configuración (ni reinicia el ESP32)
- Purgar el buffer, de modo que cualquier evento pendiente sea escrito, para esto llamamos a la RPC `FileLog.Flush`
  - por ejemplo: `mos call FileLog.Flush`
- Operar sobre los logs
- Habilitar nuevamente el logging
  - por ejemplo: `mos call Config.Set '{"config": {"file_logger": {"enable": true } }, "reboot": false, "save": false}'`

Para operar sobre los logs, el fabricante recomienda traerlos en orden de creación (el más antiguo primero) e ir borrándolos a medida que se los trae. El script de ejemplo[2] que provee Mongoose-OS, realiza esta actividad valiéndose de la utilidad *jq* que permite parsear JSON dentro de un shell script:

- Obtener el nombre del archivo más antiguo
  - por ejemplo: `mos call FileLog.Status` y pasárselo a `jq -r .oldest`
- Traerlo por partes
  - por ejemplo: `mos get --chunk-size=2048 nombredellog > destino`
- Borrarlo
  - por ejemplo: `mos rm nombredellog`
- Repetir hasta que no haya más archivos

Tengamos presente que seteando la variable de entorno *MOS_PORT* podemos indicar la forma de acceso al dispositivo sin necesidad de incluirlo en el script (aunque nada impide pasarlo como parámetro).

Finalmente, obtendremos un archivo similar a éste:

```
init.js:25              ### init script started (tambien) ###
init.js:11              Este WARN  va adentro del log
init.js:12              Este INFO tambien
init.js:16              Este WARN tambien
```

Para casos simples como este ejemplo, podemos tomar un atajo y leer directamente el archivo ejecutando `mos –-port http://192.168.4.1/rpc get <nombre del archivo>`

Para mayor información sobre cómo usar las RPC, diversas formas de acceso y autenticación, solicite a su vendedor los Comentarios Técnicos Cika correspondientes, u obténgalos de la página web de Cika.

## Tiempo del evento y tiempo real

2 https://github.com/mongoose-os-libs/file-logger/blob/master/tools/fetch_logs.sh

Si lo configuramos, cada evento se guarda con una columna adicional con un *timestamp*; un número relacionado al momento en el que se produjo dicho evento. Este número corresponde al *uptime*, es decir, al tiempo transcurrido desde el último inicio del ESP32 con Mongoose-OS, en microsegundos.
Tengamos en cuenta que el *uptime*:

- puede llegar a repetirse
- no indica información fiable de tiempo real
- no es confiable para inferir intervalos pues la secuencia se reinicia en un reset
- permite detectar reinicios observando la secuencia esperable (se produce un salto hacia atrás); pero esto puede no darse de por sí, dependiendo de cómo realizamos el logging y cómo se den los eventos. Es decir, puede haber un reinicio no detectable si no hubo nada que guardar en el intervalo, una ruptura de secuencia indica un reinicio pero una secuencia creciente no implica en sí misma que no haya ocurrido.[3]

Si además configuramos nuestro sistema para obtener fecha y hora mediante SNTP (tema que abordamos en CTC-097), el file logger agrega en esa misma columna una información adicional que permite relacionar el uptime con el tiempo real.
Esto resuelve todos los aspectos mencionados, pero requiere procesamiento del archivo, es decir, no se observa mirando el archivo a simple vista.
Configuramos la facilidad de log con *timestamps* de la siguiente forma:

```
config_schema:
  - ["file_logger.timestamps", true]  # Agrega timestamps (uptime)
```

El siguiente es un log con *timestamps*, obsérvese la ruptura de la secuencia en el reinicio forzado:

```
1464741 init.js:27              ### init script started (tambien) ###
10785403 init.js:12              Este WARN  va adentro del log
10797613 init.js:13              Este INFO tambien
13780416 init.js:17              Este WARN tambien
1461469 init.js:27              ### init script started (tambien) ###
```

El siguiente es un log con *timestamps* con el sistema configurado para sincronizar su reloj mediante SNTP, obsérvese como el sincronismo ocurre luego del mensaje de inicio, y se agrega al primer log que ocurre posteriormente:

```
1461469 init.js:27              ### init script started (tambien) ###
10494473:1591136579461359 init.js:12              Este WARN  va adentro del log
10505796 init.js:13              Este INFO tambien
14601632 init.js:17              Este WARN tambien
```

El número agregado luego de los dos puntos corresponde al tiempo UTC en microsegundos desde la *epoch* Unix[4], lo cual permite convertirlo a fecha y hora con cualquier función ad hoc.

3 Podríamos pensar entonces en forzarlo incluyendo un mensaje que se guarda al inicio del sistema, pero esto desarticularía la utilidad mencionada, que pasaría a pertenecerle a dicho mensaje...
4 La *epoch* Unix corresponde al 1 de enero de 1970. El valor devuelto es la cantidad de microsegundos transcurridos desde el comienzo de ese día.